Titre: Title:	Plate-forme de communication pour systèmes embarqués multipuces utilisant HyperTransport
Auteur: Author:	Ami Castonguay
Date:	2006
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Castonguay, A. (2006). Plate-forme de communication pour systèmes embarqués multipuces utilisant HyperTransport [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7868/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/7868/
Directeurs de recherche: Advisors:	
Programme: Program:	Non spécifié

UNIVERSITÉ DE MONTRÉAL

PLATE-FORME DE COMMUNICATION POUR SYSTÈMES EMBARQUÉS MULTIPUCES UTILISANT HYPERTRANSPORT

AMI CASTONGUAY
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)

AOÛT 2006

Your file Votre référence
ISBN: 978-0-494-19287-0
Our file Notre référence
ISBN: 978-0-494-19287-0

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

PLATE-FORME DE COMMUNICATION POUR SYSTÈMES EMBARQUÉS MULTIPUCES UTILISANT HYPERTRANSPORT

présenté par : CASTONGUAY Ami

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M^me^ NICOLESCU Gabriela, Ph.D., présidente

M. SAVARIA Yvon, Ph.D., membre et directeur de recherche

M. DAVID Jean-Pierre, Ph.D., membre

Remerciements

Ce travail a été accompli grâce aux contributions de nombreuses personnes et organismes :

Yvon Savaria – Tout d'abord un grand merci à mon directeur pour m'avoir proposé un projet qui a su me passionner tout au long de ma maîtrise. Sa vision et ses conseils m'ont toujours gardé dans le droit chemin afin d'atteindre les objectifs visés. Malgré son horaire chargé, il a toujours su être présent lorsque nécessaire.

Bernard Racine – Ses idées à l'avant-garde de la technologie ont permis de donner un sens à cette recherche. Il a su partager sa passion pour le sujet, tout en donnant de précieux conseils.

Conseil de recherches en sciences naturelles et en génie du Canada – Pour son grand appui financier grâce à son programme de bourses d'études supérieures.

CMC Microsystèmes – Pour l'accès à tous les outils de microélectronique qui ont permis de compléter le présent mémoire.

Francis St-Pierre – L'intégration de son réseau embarqué avec le tunnel HyperTransport présenté dans ce mémoire a permis de donner une dimension supplémentaire très intéressante au projet.

Jean-François Bélanger, Laurent Aubray, Michel Morneau, Max-Élie Salomon et Martin Chabot-Corriveau – Ils ont tous participé à la conception d'une première implémentation du tunnel HyperTransport. Ce fut un énorme coup de pouce!

Résumé

Même s'il est possible d'intégrer une multitude de fonctionnalités à l'intérieur d'un système embarqué sur puce, il est parfois nécessaire de distribuer ces fonctionnalités sur plusieurs puces à cause de limites technologiques ou afin d'offrir une plus grande flexibilité au système. Pour qu'un environnement multipuce fonctionne adéquatement, les composants embarqués sur différentes puces doivent pouvoir communiquer entre eux de façon performante. Ce mémoire explore la conception d'une plate-forme de communication pour de tels systèmes embarqués sur plusieurs puces.

En premier lieu, il est important de bien connaître les besoins de notre plate-forme : il est nécessaire d'avoir accès à un protocole de communication interpuces qui est à la fois performant et compatible avec les méthodes de communication actuelles des systèmes embarqués. Les protocoles de communications PCI Express, RapidIO, HyperTransport et SPI 4-2 sont comparés en terme de fonctionnalités et c'est le protocole HyperTransport qui est retenu pour la plate-forme.

Afin d'avoir accès à cette technologie de communication interpuces, la conception d'un tunnel HyperTransport a été effectuée. L'architecture et la méthode de développement du tunnel sont présentées en détail. Le tunnel a été conçu à l'aide de SystemC, à la fois pour la validation de l'architecture, la conception matérielle et pour la vérification de son bon fonctionnement. Le tunnel, qui est disponible sous une licence libre, est grandement paramétrable et supporte plusieurs fonctionnalités dont le mode *retry* qui permet de corriger des erreurs de transmission.

Les résultats de synthèse du tunnel permettent tout d'abord d'obtenir une compréhension d'où vient la grande complexité des contrôleurs de communication de haute performance. La complexité logique du tunnel se situe à 131,9 milliers de portes logiques, ce qui exclut les 43,2 Kibits de mémoires embarquées qui sont également

nécessaires. Le tunnel peut être implémenté à la fois pour la technologie de cellules normalisées TSMC 0,18 µm pour obtenir une bande passante de 8000 Mb/s et pour des FPGAs tels que ceux d'Altera et de Xilinx avec une bande passante de 2400 Mb/s et 3200 Mb/s respectivement. L'aspect paramétrable du tunnel permet également de comprendre les effets des divers paramètres du circuit sur sa complexité et sa performance.

Un pont a été conçu afin de permettre à des systèmes embarqués d'interagir avec un réseau HyperTransport. Son architecture et les résultats de synthèse sont également détaillés dans le mémoire. Ce pont a permis d'intégrer, en simulation, plusieurs réseaux embarqués RoC avec une chaîne de communication HyperTransport constitués de deux tunnels HyperTransport. La simulation d'une chaîne de communication HyperTransport a nécessité la conception d'un hôte HyperTransport, qui a été simulée à l'aide d'un modèle transactionnel en SystemC.

Abstract

Even if it is possible to integrate a great deal of functionalities inside a system on a chip, it is sometimes necessary to distribute the functionalities on multiple chips because of technological limitations or in order to achieve greater system flexibility. To obtain adequate system performance in a multiple chip environment, high performance communication is required. This thesis explores the development of a communication platform for such systems on multiple chips.

It is first necessary to fully understand all the needs of the communication platform: a high performance interchip communication protocol is needed that is compatible with existing system on chip communication. PCI Express, RapidIO, HyperTransport and SPI 4-2 are compared in terms of functionalities and the HyperTransport protocol was chosen for the platform.

In order to obtain such an interchip communication technology, a HyperTransport tunnel was developed. The tunnel architecture and the development method that was used are presented in details. SystemC was used for the validation of the architecture of the tunnel, the hardware design and to verify that all the functionalities were correct. The tunnel, that is available under an open license, is highly configurable and supports many functionalities, such as the *retry* mode that allows error correction.

Synthesis results provide a better understanding of the complexity of high performance communication controllers. The tunnel has a complexity of 131 900 logic gates, which excludes the 43,2 Kibits of embedded memory that are also required. The tunnel can be implemented in a TSMC 0,18 µm technology to obtain a bandwith of 8000 Mb/s and for FPGAs such as those from Altera and Xilinx with a resulting bandwidth of 2400 Mb/s and 3200 Mb/s respectively. The effects of the various parameters on the circuit complexity and performance are also explored in details.

A bridge was developed to allow system on chips to communicate with the HyperTransport network. The architecture and synthesis results of the bridge are also presented in details. The bridge allowed the integration of many RoC network on chips with a HyperTransport communication chain composed of two HyperTransport tunnels. Such a simulation also required a HyperTransport host controller that was simulated using a SystemC transactional model.

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et des abréviations

CMOS	Semi-conducteur à oxyde de métal complémentaire
CRC	Contrôle par redondance cyclique
FIFO	« *First In First Out* » (Premier entré, premier sorti)
FPGA	Réseau prédiffusé programmable par l'utilisateur
HT	HyperTransport
LVDS	Signalisation différentielle à basse tension
MÉF	Machine à états finis
MPL	Licence publique Mozilla
NOP	Paquet qui n'effectue aucune opération.
PCI	« *Peripheral Component Interconnect* »
RE	Réseau embarqué
REC	Registres d'espace de configuration (module du tunnel HyperTransport)
RoC	« *Rotator on chip* » (type de réseau embarqué)
RX	Réception
SoC	« *System on a chip* » (système embarqué sur une puce)
SPI	« *System Packet Interface* » (protocole de communication interpuces)

TSMC	Taiwan Semiconductor Manufacturing Company Ltd.
TX	Transmission
VHDL	« *Very-High-Speed Integrated Circuit Hardware Description Language* »

Introduction

La capacité de calcul des systèmes électroniques a vu une augmentation fulgurante au cours de la dernière décennie. Cette augmentation de capacité de calcul est le résultat de l'amélioration des procédés de fabrication, ce qui permet un accroissement de la complexité des circuits intégrés. La technologie a toutefois ses limites et, pour obtenir une performance et un coût adéquats, il est parfois nécessaire de partitionner les fonctionnalités d'un système sur plusieurs composants. Cela permet de faire des circuits plus petits qui ont un taux de rejet plus faible lors de la fabrication, en plus de permettre d'obtenir un système plus extensible en permettant de faire varier le nombre de composants inclus dans le système. Distribuer un calcul sur plusieurs puces nécessite implicitement que les divers composants communiquent entre eux. Le présent mémoire de maîtrise présente une architecture de communication qui permet à de tels composants embarqués sur plusieurs puces de communiquer entre eux de façon efficace.

Il existe présentement plusieurs protocoles de communication interpuces de haute performance dont HyperTransport [16], PCI Express, RapidIO [28] et System Packet Interface (SPI) [25]. HyperTransport (HT) a été retenu comme protocole convenant le mieux au besoin du projet pour des raisons qui seront expliquées plus en détail dans la suite de ce document. HT est un protocole par paquet de grande performance et à faible latence. Des circuits HT commerciaux sont disponibles mais ces derniers sont coûteux et laissent peu de flexibilité pour répondre aux différents besoins d'une grande variété d'applications. Une première étape du projet de recherche fut donc de faire le développement d'un circuit de contrôle HT, une étape qui a occupé une majeure partie du projet à cause de la grande complexité du protocole.

Puisque HT est un protocole de communication par paquet, il faut divers types de composants pour diriger les paquets à l'intérieur d'un système incluant des routeurs. Un

routeur est un composant qui comprend plus de deux ports HT et qui permet de diriger les paquets d'un port à un autre. Puisqu'un routeur est un élément complexe, débuter par faire la conception d'un tunnel était préférable, puisque c'est un élément de base plus simple. Un tunnel HT est similaire à un routeur HT, mais celui-ci ne comporte que deux liens. Il est donc beaucoup plus simple à concevoir qu'un routeur, tout en comportant une série de composants qui pourraient être réutilisés pour faire la conception d'un routeur. Une part importante de ce document porte donc sur les détails de conception de ce circuit ainsi que sur les résultats obtenus qui permettent une analyse approfondie de la complexité d'un tel circuit. Les données détaillées nécessaires à une telle analyse n'étaient jusqu'à maintenant pas disponibles dans la littérature. Afin de rendre le plus disponible possible les résultats de notre recherche, le tunnel a été rendu librement disponible sous la licence MPL (Mozilla Public Licence) [22].

Un aspect intéressant de la conception du tunnel est qu'il a été développé en utilisant exclusivement SystemC [32], à la fois pour la vérification et la synthèse. SystemC est une bibliothèque C++ qui permet de modéliser le comportement d'un circuit à divers niveaux d'abstraction. Cette capacité du C++ de définir la fonctionnalité d'un circuit à haut niveau d'abstraction permet de valider rapidement en simulation le bon fonctionnement de son architecture. Des outils de synthèse de SystemC sont également disponibles et ceux-ci permettent de rapidement prototyper un circuit sans devoir utiliser un langage de synthèse plus traditionnel tel que le VHDL ou le Verilog. L'utilisation d'un tel flot de conception comporte des avantages pour permettre de rendre paramétrable le circuit, mais comporte aussi des désavantages, puisque c'est un flot de conception peu conventionnel.

L'effort de développement nécessaire pour concevoir des circuits numériques intégrés est en constante croissance à cause de leur complexité grandissante. Une solution pour limiter l'ampleur de la tâche de conception est de réutiliser des circuits et de les intégrer dans un système embarqué qui regroupe une multitude de composants. La

réutilisation de circuits permet de faire la conception d'un système avec des ressources de développement limitées.

Tout comme pour un système comprenant plusieurs puces, un système embarqué sur une puce nécessite une architecture de communication permettant à ses divers composants d'échanger de l'information. Si l'on veut créer un système embarqué extensible, le système de communication doit être en mesure de communiquer avec d'autres puces. La méthode proposée dans ce mémoire pour effectuer cette tâche est d'intégrer à un système embarqué la capacité de communiquer par un protocole interpuces de haute performance. HT est par contre un protocole complexe et les composants d'un système embarqué sont souvent conçus pour communiquer avec les autres composants à l'aide de protocoles beaucoup plus simples. Pour que tous les éléments d'un système embarqué puissent profiter de communication interpuces, il faut donc un pont pour effectuer une traduction de la communication entre HT et le système embarqué.

La création d'un pont pour lier un réseau embarqué avec le réseau HT est donc une étape afin de permettre à une multitude de puces contenant des systèmes embarqués de communiquer entre elles. L'intégration grâce à un réseau HT d'un système embarqué comprenant divers éléments ainsi que des ponts pour lier tous ces éléments permet donc d'obtenir une architecture de communication pour système embarqué sur plusieurs puces, dont on peut ensuite vérifier le bon fonctionnement à l'aide de simulations.

Contributions du présent mémoire

Dans un premier temps, le présent projet donne à la communauté scientifique accès à un tunnel HyperTransport, puisque celui-ci est librement disponible. Ce tunnel pourra être modifié et réutilisé pour des projets nécessitant des liens interpuces performants. Le développement de ce tunnel permet également de recueillir quantité de données sur la conception de contrôleur pour lien interpuces de haute performance. Il permet

d'approfondir les connaissances sur la complexité de tels contrôleurs puisqu'aucune donnée sur le sujet n'est disponible dans la littérature. L'utilisation du langage SystemC pour faire la conception du tunnel permet en plus de faire la mise au point sur l'état des outils de synthèse de SystemC, un langage qui est encore très peu utilisé pour cette tâche.

D'autre part, la partition d'une application embarquée entre plusieurs puces n'est pas un aspect nouveau [33], mais la méthode à utiliser pour effectuer la communication entre plusieurs systèmes embarqués distribués sur plusieurs puces l'est. Des systèmes embarqués peuvent souvent inclure des liens de communication interpuces mais la solution présentée est innovatrice du fait qu'elle s'intègre de façon entièrement transparente au système de façon à ce qu'une communication intra puce s'effectue de façon identique à une communication hors puce.

Plan du mémoire

Le mémoire débute par faire une revue de littérature pour bien établir le contexte présent du sujet de recherche. On y retrouve une discussion sur une variété de thèmes abordés par le projet, de la communication interpuces jusqu'aux systèmes embarqués, en passant par les méthodes de conception en utilisant SystemC.

Le chapitre 2 continue en présentant l'architecture d'un système de calcul distribué sur plusieurs puces, la raison pour laquelle le projet a pris forme. On y fait le choix d'un protocole de communication interpuces et on y présente comment celui-ci peut interagir avec des systèmes embarqués.

Le chapitre 3 aborde pour sa part les détails techniques de la conception du tunnel HT. On débute en présentant la licence qui couvre le tunnel HT. On voit ensuite un survol de l'architecture, la méthode de conception qui a été utilisée et les éléments paramétrables du tunnel. Le chapitre se termine avec une présentation détaillée de

l'architecture du tunnel. La présentation du tunnel se poursuit ensuite au chapitre 4 où les résultats de synthèse du tunnel sont présentés.

Le chapitre 5 décrit l'intégration du tunnel avec un système embarqué à l'aide du pont. On y présente donc le pont, utilisé pour lier HT et le tunnel, ainsi qu'un hôte HT qui permet de contrôler la chaîne de communication HT.

Chapitre 1. Revue de littérature

Le développement de systèmes de communication est un aspect extrêmement important pour une multitude d'applications et il est possible de trouver dans la littérature une variété d'informations dans ce domaine. Dans le cadre de ce projet, quatre grands sujets attirent notre attention : les technologies de communication interpuces, HyperTransport plus spécifiquement, les méthodes de développement de circuits en utilisant le langage SystemC et finalement les réseaux embarqués et la façon dont ils peuvent s'intégrer à un tissu de communication interpuces.

1.1 Communication interpuces

Une première étape pour permettre de bien comprendre le domaine de la communication interpuces est de connaître les différents grands joueurs de ce domaine. HyperTransport, RapidIO, PCI-Express et System Packet Interface (SPI) sont quatre protocoles dominants présentement. Toutes les spécifications de ces protocoles ont été créées par des regroupements de sociétés : le consortium HyperTransport pour HyperTransport, RapidIO Trade Association pour RapidIO, PCI-SIG pour PCI-Express et Optical Internetworking Forum pour SPI. Il peut également être pertinent de mentionner la technologie FlexIO™ de Rambus pour laquelle peu d'information est librement disponible, ainsi que Cray qui utilise dans ses superordinateurs une combinaison de HyperTransport et de protocoles propriétaires pour permettre à des milliers de processeurs de communiquer entre eux à l'intérieur d'une matrice de communication tridimensionnelle [5].

Tous ces protocoles utilisent des liens électriques différentiels qui comportent deux fils sur lesquels on envoie une tension différentielle. Ce mode de transmission permet une plus grande immunité au bruit que l'utilisation d'un fil unique et permet donc d'utiliser des tensions plus basses et d'atteindre des fréquences de transmission plus

élevées. Un aspect également très important de ces protocoles est qu'ils utilisent tous deux groupes indépendants de signaux pour la transmission et la réception d'information, permettant donc d'envoyer et de recevoir de l'information de façon simultanée. Les spécifications électriques de ces liens varient par contre d'un protocole à un autre. PCI Express et RapidIO utilisent tous deux des liens sériels utilisant un encodage qui intègre l'horloge et les données dans un même flot de données tandis que HyperTransport et SPI utilisent des liens parallèles avec des signaux de contrôle et d'horloge indépendants des signaux de transfert de données. L'utilisation d'un lien sériel occasionne un surdébit par rapport à l'utilisation d'un lien parallèle, mais elle permet d'utiliser des lignes de transmission plus longues. Un aspect extrêmement important de tous ces protocoles est qu'ils effectuent tous du contrôle de flux de façon matérielle. Le contrôle de flux est nécessaire afin de s'assurer que la destination d'un paquet est en mesure de le recevoir et de le traiter sans perte de données.

Du côté des fonctionnalités, HyperTransport, RapidIO et PCI Express sont tous relativement similaires. Ces trois protocoles définissent des paquets et effectuent de la détection et de la correction d'erreurs si désiré. Un paquet consiste en un en-tête qui contient la destination du paquet ainsi que d'autres informations de contrôle suivi de données. Un paquet peut donc être acheminé à l'intérieur d'un réseau de puces jusqu'à une destination bien précise. Pour tous ces protocoles, les paquets peuvent passer à l'intérieur de canaux virtuels indépendants, évitant ainsi que certains types de paquets empêchent d'autres types de paquets de passer, ce qui permet d'éviter des interblocages ou d'améliorer la performance pour acheminer des paquets prioritaires. Par contre, HyperTransport et PCI Express utilisent un système de mémoire partagée afin de diriger les paquets à l'intérieur d'un système tandis que RapidIO définit une couche de transport qui fait complètement abstraction du contenu du paquet. De son côté, SPI est un protocole plus simple qui permet d'envoyer des données à une destination précise, sans définir des types de paquets spécifiques. SPI est bien adapté pour transmettre un flux de

données vers des destinations, puisqu'il ne comprend qu'un canal de transmission pour chaque destination. Une comparaison détaillée de ces protocoles est effectuée par [6] et est reprise au tableau 1 avec l'ajout du protocole SPI-4.2.

Tableau 1 - Comparaison des protocoles de communication interpuces PCI-Express, RapidIO sériel, Hypertransport et SPI 4-2

	PCI Express	RapidIO Sériel	HyperTransport	SPI-4.2
Couche physique	• Lien LVDS sériel (2.5 GHz) • 1, 2, 4, 8, 16 ou 32 liens • Compensation et récupération d'horloge • Séparation en couloirs et correction d'alignement • Codage 8/10b • Verrouillage de trame : Code de début/fin et codes de commande • Brouillage pour réduire les émissions électromagnétiques	• Lien LVDS sériel (1.25, 2,5 ou 3,125 GHz) • 1 ou 4 liens • Compensation et récupération d'horloge • Séparation en couloirs et correction d'alignement • Codage 8/10b • Verrouillage de trame : Code de début et codes de commande	• Lien LVDS parallèle (0,4 – 2,4 GHz) • 2 – 32 données, 1 contrôle, 1-4 horloge • Compensation d'horloge • Séparation en couloirs et correction d'alignement • Pas de codage • Pas de verrouillage de trame	• Lien LVDS parallèle (0,311 GHz et plus) • 16 données, 1 contrôle, 1 horloge et 2 données et 1 horloge pour statut de contrôle de flux • Compensation d'horloge • Pas de couloir • Pas de codage • Pas de verrouillage de trame
Couche de lien	• Protection de CRC 16 bits par paquet de lien et CRC de 32 bits pour la communication bout à bout • Protocole Ack/Nack pour chaque lien • Classification de type de paquet et assemblage • Paquets de lien de données de 6 octets	• Protection de CRC 16 bits par paquet de lien et CRC de 5 bits pour les codes de contrôle • Protocole Ack/Nack pour chaque lien • Classification de type de paquet et assemblage • Paquets de lien de données de 3 octets	• Protection de CRC 32 bits pour chaque tranche de 512 octets transmis (détection erreur) et protection 32 bits par paquet optionnelle • Protocole Ack/*retry* optionnel • Classification de type de paquet (pas d'assemblage : la taille des paquets est connue) • Pas de paquet de lien	• Protection par parité diagonale entrelacée de 4 bits pour les données et 2 bits pour le contrôle de flux • Pas de correction d'erreur • Pas de classification puisqu'il n'y a pas de type de paquet, assemblage • Pas de paquet de lien
Couche transport	• Trois types de trafic : posted, non-posted et completion, avec des canaux individuels • 8 priorités de trafic à l'aide de canaux virtuels • Transferts de données jusqu'à 4096 octets, entête typique de 12 à 16 octets • Contrôle de flux par crédit. Les crédits sont gérés par type de trafic. • Jusqu'à 32 transactions en cours • Espace de configuration	• Pas de canal individuel par type de trafic • 4 priorités de trafic avec des canaux indépendants • Transferts de données jusqu'à 256 octets. En-tête typique de 6 octets • Contrôle de flux au niveau de la couche physique par essais successifs ou par crédit • Jusqu'à 256 transactions en cours, dont 32 non confirmées • Espace de configuration	• Trois types de trafic : posted, non-posted et response, avec des canaux individuels • Possibilité de relaxer les règles de réorganisation et un deuxième niveau de priorité optionnel (isoc) • Transferts de données jusqu'à 64 octets, en-tête typique de 4 à 8 octets • Contrôle de flux par crédit. Les crédits sont gérés par type de trafic. • Jusqu'à 32 transactions en cours par UnitID : un noeud peut avoir plus d'un UnitID. • Espace de configuration	• Pas de priorité de trafic • Transfert de données maximal déterminé par l'application, en-tête de 2 octets • Contrôle de flux par combinaison d'échange de statut de tampons et de crédits

HyperTransport est le bus de communication utilisé par les processeurs les plus récents de la société AMD, ce qui veut dire qu'il se retrouve dans une grande quantité d'ordinateurs et occupe donc une bonne part du marché. HT est également utilisé sur les cartes mères pour permettre aux différentes puces d'un jeu de puces de communiquer entres elles. Il est également utilisé par des processeurs réseaux. Jusqu'à tout dernièrement, HT ne pouvait pas servir pour la mise en œuvre de fond de panier (*backplane* : ensemble de connecteurs permettant l'ajout de cartes amovibles) puisqu'aucun connecteur HT n'était défini, mais cela vient tout récemment de changer avec l'introduction de la norme HTX [17]. HTX utilise le même connecteur physique que PCI Express, mais orienté différemment afin d'éviter de placer une carte PCI Express dans ce connecteur. RapidIO de son côté a été fortement adopté par les processeurs de traitement de signal en plus de servir également pour des processeurs de réseau par Motorola, l'entreprise qui en est l'instigatrice. PCI Express est un joueur beaucoup plus récent mais qui a su prendre de l'ampleur très rapidement puisque ce protocole est devenu la norme pour les cartes vidéo d'ordinateurs personnels. Son succès est lié au fait qu'il est supporté par Intel qui est un joueur de taille dans l'industrie des ordinateurs personnels. De son côté, SPI est utilisé dans les commutateurs réseaux de grande performance.

Pour ce qui est de faire la conception de circuits pour utiliser ces divers protocoles, aucune information n'est facilement disponible à ce sujet. Il existe toutefois des circuits que l'on peut acheter, pour utilisation à l'intérieur d'un FPGA, qui fournissent des spécifications expliquant les grandes lignes de leur architecture et la quantité de ressources qu'ils requièrent. Ces spécifications ne donnent par contre pas de détails au sujet de la distribution de la complexité. Il y a donc un intérêt de déterminer d'où vient la complexité des circuits utilisés pour ces protocoles.

1.2 HyperTransport

Suite à une analyse qui sera élaborée avec plus de détails au cours de ce document, la technologie de communication interpuces HyperTransport [16] a été retenue pour effectuer la communication interpuces au sein de notre système. HT est un protocole de communication interpuces qui permet à une grande variété de composants électroniques d'échanger de l'information de façon performante. La topologie de base d'un réseau HT est une chaîne de composants qui sont reliés par des connexions point à point, tel qu'il est démontré à la figure 1.1. Une chaîne comporte toujours un hôte, un élément de fin de chaîne nommé « Cave » et un nombre variable de tunnels qui comportent chacun deux liens de communication HT. Une topologie en étoile peut également être utilisée grâce à l'utilisation de commutateurs HT.

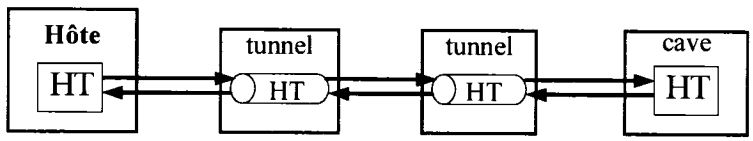

Figure 1.1 - Topologie d'une chaîne HT

1.2.1 Lien physique et initialisation

Le lien électrique de HyperTransport comprend plusieurs lignes permettant de transmettre des données, un signal horloge et un signal de contrôle, et cela pour la transmission et la réception de données. Chaque ligne est un lien différentiel LVDS offrant une grande résistance au bruit électromagnétique tout en ayant une faible consommation énergétique. Le nombre de lignes de données peut varier de 2 à 32 en fonction de la bande passante requise, et cela pour la transmission et la réception. Les données sont transmises aux deux fronts du signal d'horloge. Le récepteur doit se baser sur le signal d'horloge reçu afin de faire la lecture des données.

Les signaux sont regroupés en couloir qui sont composés d'un maximum de 8 signaux de données, un signal de contrôle et un signal d'horloge. Toutes les lignes d'un

couloir doivent avoir la même longueur, afin que les signaux électriques prennent le même temps à atteindre la destination et demeurent ainsi le plus possible en phase. La séparation des signaux en couloirs facilite la conception de circuits imprimés puisqu'il n'est pas nécessaire que toutes les lignes du lien aient la même longueur.

Chaque couloir doit être initialisé individuellement à l'aide d'une séquence d'initialisation. Cette séquence permet de s'assurer qu'à la fois le récepteur et le transmetteur sont prêts à débuter la communication et sert également à indiquer à quel moment les paquets commencent à être transmis.

1.2.2 Format des paquets

L'information transmise par HT transite par divers types de messages qui incluent des écritures, des lectures et des réponses. Un système d'adressage de 40 bits ou de 64 bits, au choix du concepteur, est utilisé pour acheminer les paquets à leur destination. HT est un protocole de type stockage et retransmission : chaque nœud HT contient trois groupes indépendants de tampons qui permettent d'accumuler divers types de paquets :

- *Posted* – requête qui ne requiert pas de réponse, telle qu'une écriture.
- *Non-Posted* – requête qui requiert une réponse, telle qu'une lecture.
- *Response* – réponse à une requête, telle qu'une réponse à une lecture.

L'en-tête qui contient toutes les informations sur le type de paquet se nomme un paquet de commande, tandis que les données associées à l'opération, s'il y en a, se nomment un paquet de données. Il existe quatre grandes familles de paquets : *Info*, *Request*, *Address Extension* et *Response*.

Les paquets de la famille *Info* ont une taille de 4 octets et ne servent qu'à la communication entre deux voisins. Cette catégorie comprend le paquet *NOP* qui est

transmis afin d'échanger de l'information sur l'espace tampon disponible ou lorsqu'aucun autre paquet ne peut être transmis. Il comprend également le paquet *Sync* qui est envoyé lorsqu'une erreur critique est détectée et que la chaîne entière doit être réinitialisée.

Le support pour les paquets de la famille *Address Extension* est optionnel et permet d'ajouter, à certains paquets de requête, 24 bits supplémentaires d'adresse pour obtenir une adresse de 64 bits au lieu de 40 bits.

Les paquets de la famille *Request* ont une taille de 4 ou 8 octets et permettent d'effectuer une requête. Cette famille comprend les paquets de type *Read*, *Write*, *Atomic-RMW*, *Broadcast*, *Flush* et *Fence*. Les paquets de type *Read*, *Write* et *Atomic-RMW* permettent d'accéder et/ou de modifier le contenu d'un espace mémoire spécifique. *Broadcast* permet d'envoyer un message à toute la chaîne de communication HT. *Fence* permet de séparer plusieurs groupes de requêtes, puisque la majorité des requêtes n'ont pas le droit de dépasser ce type de paquet. Finalement, *Flush* permet d'obtenir une réponse lorsque toutes les requêtes de type *posted* envoyées précédemment vers l'hôte ont atteint ce dernier.

Tableau 2 - Format de la famille de paquet *Request*

Octet	7	6	5	4	3	2	1	0
0	SeqID[3:2]		Cmd[5:0]					
1	PassPW	SeqID[1:0]		UnitID[4:0]				
2	Spécifique à Cmd							
3	Spécifique à Cmd							
4	Addr[15:8]							
5	Addr[23:16]							
6	Addr[31:24]							
7	Addr[39:32]							

Les paquets de la famille *Request* ont tous le format présenté au tableau 2. Le champ *Cmd* détermine le type exact du paquet tandis que les champs spécifiques à *Cmd* ont une fonctionnalité qui varie en fonction du type exact du paquet. *SeqID* détermine si ce paquet fait partie d'une séquence. Des paquets d'une même séquence doivent absolument être traités dans l'ordre dans lequel ils sont transmis. *UnitID* permet de déterminer quel est l'émetteur du paquet afin de permettre à un paquet de réponse de revenir à celui qui a émis la requête. Finalement, le champ *PassPW* détermine si ce paquet a le droit de dépasser les paquets de type *Posted Write*, ce qui permet d'assouplir les règles de maintien de l'ordre des paquets afin d'améliorer les performances du système.

Les paquets qui jouent le rôle d'entête pour des données, tel qu'un paquet *Write* ou *Atomic-RMW,* contiennent tous deux un champ *Count* qui détermine la taille, en nombre de double mot (32 bits), du paquet de données qui suit la requête. Les paquets qui doivent effectuer une opération à une adresse spécifique contiennent le champ *Addr* qui se trouve à être l'adresse ou doit s'effectuer l'opération. C'est le champ *Addr* qui détermine la destination du paquet. Il est important de noter que plusieurs plages d'adresses sont réservées afin de permettre d'envoyer des paquets spécifiquement à un élément de la chaîne sans devoir nécessairement connaître la plage mémoire utilisée par cet élément (*Device Messaging*), d'envoyer des interruptions et de supporter plusieurs autres fonctionnalités. Les paquets pour lesquels une réponse doit être générée lorsqu'arrivés à destination contiennent également un champ *SrcTag* qui est un identificateur unique permettant d'associer la réponse à la bonne requête.

Les paquets de la famille *Response* sont générés en réponse à une requête et ont toujours une taille de 4 octets. Cela peut être le résultat de l'opération *Read* et *Atomic-RMW*, ou pour avertir qu'une opération tel qu'un *Write* ou *Flush* est complétée avec succès.

Tableau 3 - Format de la famille de paquet *Response*

Octet	7	6	5	4	3	2	1	0
0	Spécifique à Cmd		Cmd[5:0]					
1	Spécifique à Cmd		Rsv	UnitID[4:0]				
2	PassPW	Bridge	Error0	SrcTag[4:0]				
3	Rsv/RqUID		Error1	Fonctionnalité avancée			Spécifique à Cmd	

Les paquets de la famille *Response* ont tous le format présenté au tableau 3. Le champ *Cmd* détermine le type exact du paquet tandis que les champs spécifiques à *Cmd* ont une fonctionnalité qui varie en fonction du type exact du paquet. Les champs *UnitID* et *RqUID* permettent de déterminer la destination du paquet de réponse tandis que le champ *SrcTag* permet d'identifier à quelle requête est associée la réponse. Les bits *Error0* et *Error1* permettent d'identifier si des erreurs sont survenues lors du traitement de la requête et les bits *Rsv* sont réservés pour un usage futur.

1.2.3 Ordonnancement des paquets

Puisque HyperTransport comporte trois canaux virtuels indépendants et afin d'optimiser la performance du réseau, il est possible que certains paquets dépassent d'autres paquets lorsqu'un canal virtuel est bloqué. Afin de bien comprendre les règles régissant la réorganisation des paquets, il faut savoir que deux paquets sont considérés dans le même flot lorsqu'ils sont dans la direction *downstream* ou qu'ils sont dans la direction *upstream* et qu'ils ont la même source (le même *UnitID)*. Un paquet *upstream* se dirige vers l'hôte et un paquet *downstream* s'éloigne de l'hôte. Voici les règles d'ordonnancement de HyperTransport :

1. Les paquets étant dans des flots différents peuvent être réorganisés librement.

2. Les paquets étant dans le même flot et ayant une valeur identique et non zéro du champ SeqID ne peuvent pas être réorganisés : ils font partie d'une séquence ordonnée.

3. Les paquets étant dans le même flot et ne faisant pas partie d'une séquence doivent suivre les règles du tableau 4.

Tableau 4 - Règles d'ordonnancement des paquets

Ligne peut passer colonne?	**Requête *Posted***		**Requête *Nonposted***	***Response***	
	PPW=0	**PPW =1**		**PPW =0**	**PPW =1**
Requête *Posted*, PPW=0	Non	Non	Oui	Oui	Oui
Requête *Posted*, PPW=1	Oui/Non	Non	Oui	Oui	Oui
Requête *Nonposted*, PPW=0	Non	Non	Oui/Non	Oui/Non	Oui/Non
Requête *Nonposted*, PPW=1	Oui/Non	Oui/Non	Oui/Non	Oui/Non	Oui/Non
Response, PPW=0	Non	Non	Oui	Non	Non
Response, PPW=1	Oui/Non	Oui/Non	Oui	Oui/Non	Non

Non – La transaction ne peut pas être complétée avant celle qui la précède.

Oui – La transaction doit pouvoir passer en avant de celle qui la précède si cette dernière ne peut pas être complétée (par exemple à cause d'un canal virtuel bloqué).

Oui/Non – Il n'y a pas de dépendance entre les deux transactions, mais la réorganisation des paquets n'est pas nécessaire. Une implémentation peut décider d'effectuer la réorganisation si elle y voit un avantage.

PPW – Champ *PassPW* du paquet

1.2.4 Contrôle de flux

Un paquet de données a une taille variant de 4 à 64 octets et est toujours associé à un paquet de commande, puisque ce dernier contient toute l'information sur la destination et le rôle de ces données. Le paquet de type *NOP* permet d'effectuer du contrôle de

flux puisqu'il contient des champs permettant d'échanger de l'information sur l'espace disponible dans les tampons.

Afin d'éviter le dépassement de capacité lors de la réception de paquets, HT utilise un système de crédits. Lorsqu'un espace tampon devient disponible afin de recevoir un certain type de paquet, un crédit est alloué et envoyé au nœud HT voisin. La fonctionnalité de base de HT prévoit deux types de crédits pour chaque grande catégorie de paquets : un pour les paquets de commandes et un autre pour les paquets de données. Avant d'envoyer un paquet, un émetteur doit avoir un crédit pour le type de paquet à envoyer. Dans le cas d'un paquet qui joue le rôle d'en-tête de données, il est nécessaire d'avoir à la fois un crédit pour le paquet et un autre pour le paquet de données qui suivra.

1.2.5 Vérification d'erreurs

HyperTransport prévoit deux types de vérification d'erreurs. Le premier type est une vérification de l'intégrité du lien entre deux voisins. Chaque couloir HT doit insérer périodiquement un code CRC calculé à partir des données transmises. Le récepteur effectue le même calcul de CRC sur les données reçues et le compare au code CRC reçu. Si le code CRC calculé diffère du code reçu, cela veut dire qu'au moins une erreur est survenue à l'intérieur de la fenêtre de calcul de ce code CRC. Puisqu'il n'est pas possible de corriger l'erreur, la façon dont la détection de telles erreurs est traitée est spécifique à chaque application.

Le deuxième type de vérification d'erreur se nomme le mode *retry*. Celui-ci est optionnel et permet d'assurer l'intégrité de tous les transferts de paquets. Un code CRC est inséré à la suite de chaque paquet, et chaque paquet transmis est enregistré temporairement dans une mémoire. Si le code CRC calculé à la réception n'est pas identique à celui reçu, le lien de communication est réinitialisé et le paquet est retransmis. Si aucune erreur n'est détectée, le paquet peut être retiré de la mémoire du transmetteur.

1.2.6 Configuration des fonctionnalités

Chaque élément HT comprend un espace de registres de configuration. Certains registres sont en lecture seule et permettent d'afficher les options supportées par le tunnel tandis que d'autres sont également accessibles en écriture et permettent de configurer les fonctionnalités du noeud HT. Par exemple, tous les éléments HT ont un registre qui permet de leur donner un numéro *UnitID* unique qui permet de les identifier dans la chaîne. Une plage mémoire est réservée pour l'accès à l'intérieur de cet espace de configuration. Chaque élément est responsable de détecter si un paquet reçu s'adresse à son espace de configuration et si oui, de traiter la requête adéquatement.

1.3 Systèmes et réseaux embarqués

L'amélioration des procédés de fabrication de circuits intégrés permet d'intégrer sur une même puce de plus en plus de fonctionnalités. Cette augmentation constante du niveau d'intégration a par contre l'effet d'augmenter de façon significative l'effort de développement nécessaire. Pour réduire cet effort de développement, l'industrie se penche de plus en plus sur la réutilisation de composants qui peuvent être intégrés dans ce que l'on appelle un système embarqué.

De façon traditionnelle, les composants d'un système embarqué communiquent entre eux à l'aide d'un bus. Un bus a l'avantage d'utiliser peu de ressources, mais un seul composant à la fois peut utiliser le bus pour envoyer ou recevoir de l'information. Cette méthode peut donc causer un goulot d'étranglement lorsque plusieurs composants veulent communiquer simultanément. Une méthode alternative est d'utiliser un réseau embarqué qui permet d'échanger des paquets entre les composants. Cela a l'avantage d'éliminer le goulot d'étranglement qu'est traditionnellement le bus et donc d'intégrer une plus grande quantité de composants, mais à un coût également plus élevé puisqu'il faut des éléments pour diriger les paquets jusqu'à la bonne destination.

Il existe une grande variété de topologies de réseaux embarqués. On y retrouve des réseaux en anneaux où tous les éléments d'un réseau sont placés les uns à la suite des autres, tel qu'il est proposé pour un réseau compatible à HyperTransport [30]. Les réseaux en anneaux sont très simples et ne demandent pas un grand nombre de ressources, mais ceux-ci sont peu extensibles puisque la latence à l'intérieur d'un tel réseau est proportionnelle à sa taille. Un *crossbar* tel qu'il est utilisé par le réseau *STbus crossbar* [29] permet à chaque élément de communiquer directement avec chaque autre élément du réseau. Un *crossbar* est une topologie parmi les plus performantes, puisqu'elle contient un lien direct entre chaque noeud du réseau. Cette performance vient par contre au coût d'une très grande complexité.

Certains groupes de recherche travaillent plutôt sur des réseaux en arbres où les paquets passent par des concentrateurs pour être redirigés vers la bonne destination, tels que le Butterfly Fat Tree [3] et le réseau SPIN [26]. Une matrice de noeuds est également une topologie explorée par le réseau embarqué Eclipse [20]. Ces deux derniers types de réseaux effectuent un compromis entre performance et complexité, puisqu'ils sont beaucoup plus efficaces que les anneaux, mais également beaucoup moins complexes qu'un *crossbar*.

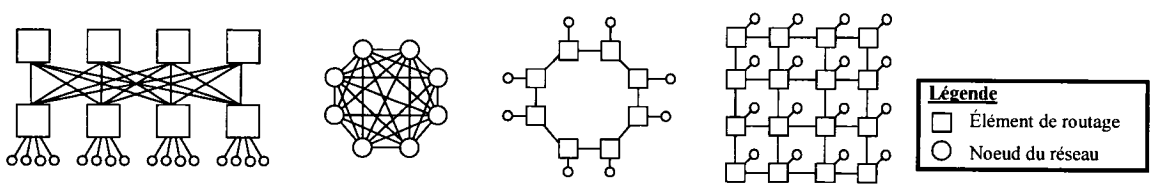

Figure 1.2 - Topologies de réseaux embarqués

Le réseau RoC [11] est un réseau embarqué développé à l'École Polytechnique en collaboration avec la société STMicroelectronics. Celui-ci combine une topologie à anneau avec la technique crossbar : chaque élément du réseau a un anneau dédié pour recevoir des paquets. La figure 1.3 montre un RoC de 4 ports. Afin d'absorber les

variations de débits des paquets dans le réseau, le RoC utilise des mémoires tampons de type premier entré, premier sorti (FIFO) lors de la transmission et de la réception de paquets. Chaque paquet transmis dans un RoC contient un champ de destination ainsi qu'un champ de données. Pour envoyer un paquet vers une destination donnée, il suffit de l'insérer dans le FIFO de transmission et le paquet sera dirigé vers le bon registre de destination par le bloc de sélection de destination. Au cours de plusieurs cycles d'horloge, les paquets sont dirigés vers la destination voulue. Bien que la figure 1.3 ne montre qu'un RoC de 4 éléments, la taille de ce réseau embarqué peut s'adapter aux besoins des applications.

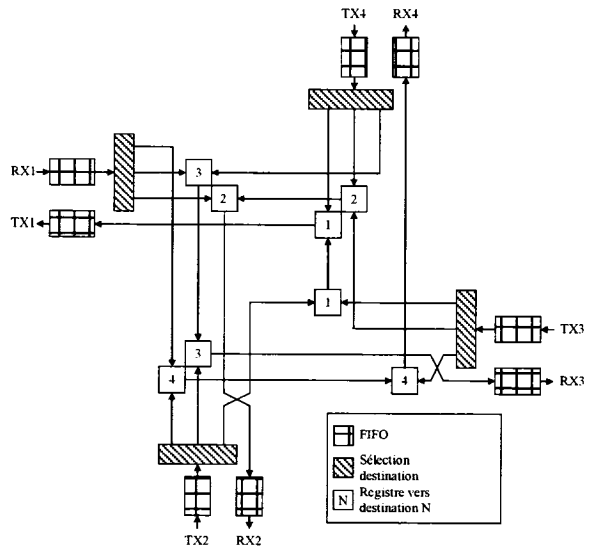

Figure 1.3 - RoC à 4 ports

Un désavantage des bus et des réseaux embarqués est qu'ils ne fournissent pas de garantie de performance, on les appelle des réseaux « *service au mieux* ». Si le bus du réseau est déjà fortement sollicité, il est possible que la performance de la communication soit très basse. Cela peut causer de graves problèmes dans des systèmes embarqués qui doivent répondre à des critères de performance très rigoureux. Une alternative à l'utilisation d'un réseau embarqué par paquets est d'utiliser un réseau embarqué avec routage par circuit [27]. Lorsqu'une route entre deux points est établie, la bande passante

et la latence entre les deux points est fixe. Cette méthode a l'avantage de garantir la performance, mais elle peut fortement limiter la flexibilité du système. Une méthode pour permettre de garantir la performance tout en gardant une certaine flexibilité est de combiner à la fois les réseaux à performance garantie ainsi que des réseaux *service au mieux* [15][9]. Certains systèmes tel que le processeur RAW [21] utilisent la tactique d'utilisation de deux réseaux distincts dont un de type *service au mieux.*

Un système embarqué doit en général être en mesure de communiquer avec le monde extérieur. La communication extérieure peut être utilisée afin de traiter des données provenant de l'extérieur, tandis que d'autres systèmes nécessitent des liens de communication externes pour partager les tâches de calcul comme pour les puces de la société AMD [4] qui peuvent communiquer avec plusieurs autres processeurs à l'aide du protocole HT. Le processeur Cell [7] est également une architecture intéressante qui combine sur une même puce un microprocesseur et 7 coprocesseurs, le tout lié par un bus interne et un lien FlexIO™ pour communiquer avec d'autres puces.

1.4 Conception matérielle utilisant SystemC

SystemC [32] est une bibliothèque C++ qui permet de modéliser le fonctionnement d'un circuit à divers niveaux d'abstraction. Il permet donc de rapidement modéliser un circuit très complexe de façon comportementale afin de valider son architecture, ou de générer des vecteurs de test pour effectuer la validation d'un circuit. Il est également possible de modéliser un circuit à un niveau d'abstraction très bas pour définir très précisément le comportement d'un circuit, tout comme il est possible de le faire avec le langage de description matérielle VHDL. Lorsque le niveau d'abstraction est suffisamment bas, il devient possible de faire la synthèse du code SystemC pour obtenir un circuit numérique.

Les circuits numériques ont la particularité de traiter toutes les données à leurs entrées en parallèle, tandis que les logiciels informatiques doivent en général effectuer

des traitements de séquentiels. SystemC permet de modéliser ce parallélisme puisqu'il comprend des structures qui permettent de définir plusieurs fils d'exécution qui doivent s'exécuter en parallèle lorsque certains évènements se produisent. Cette bibliothèque comprend également plusieurs structures qui permettent de représenter des entrées, des sorties et des signaux de circuits. SystemC intègre toute la structure nécessaire pour gérer tous les fils d'exécution et la mise à jour des signaux du système. Un exécutable produit à partir d'un code source utilisant la bibliothèque SystemC comprend donc à la fois le modèle du circuit et tout ce qui est nécessaire pour effectuer la simulation du circuit. Lors de l'exécution du modèle du circuit, il est possible de produire un fichier contenant la valeur des signaux du circuit modélisé lors de la simulation.

Ce qui suit est un compteur matériel utilisé dans le tunnel HyperTransport. Le premier fichier cd_counter_l3.h est l'en-tête du compteur. Tout composant qui utilise une instance de module n'a qu'à inclure ce fichier pour obtenir la définition de son interface. Ce même fichier inclut à son tour l'en-tête <systemc.h> qui définit la bibliothèque SystemC. Le fichier cd_counter_l3.cpp contient quant à lui l'implémentation du compteur. Un circuit modélisé en SystemC est une classe qui est dérivée de la classe sc_module. Les membres de cette classe incluent les entrées et les sorties du circuit, ainsi que ses signaux internes. Les méthodes de la classe représentent les fonctionnalités du circuit.

Fichier cd_counter_l3.h :

```
#ifndef CD_COUNTER_L3_H
#define CD_COUNTER_L3_H

//Inclusion de la bibliothèque SystemC
#include <systemc.h>

///Compteur pour le module decoder_l2
/**
        @class cd_counter_l3
        @description Compte le nombre de données reçues et active
                un signal lors de la fin de réception des données
*/

class cd_counter_l3 : public sc_module
```

```
{
        ///La valeur du compte
        sc_signal<sc_uint<4> > countVal;

public:

        //*******************************
        //      Entrées
        //*******************************

        ///Signal d'horloge de synchronisation
        sc_in< bool > clk;
        ///Signal d'initialisation (actif bas)
        sc_in< bool > resetx;
        ///Permet de fixer le nombre de données à recevoir
        sc_in< bool > setCnt;
        ///Permet de décrémenter le compte interne
        sc_in< bool > decrCnt;
        ///Paquet d'en-tête, les bits (25,22) contiennent le nombre de données qui suivent
        sc_in<sc_bv<32> > data;

        //*******************************
        //      Sorties
        //*******************************

        ///La prochaine donnée sera la dernière à recevoir
        sc_out< bool > end_of_count;

        /**
                Processus sensible au front montant de l'horloge qui compte
                Le nombre de données reçues
        */
        void count_process();

        ///Macro SystemC : la classe contient des processus
        SC_HAS_PROCESS(cd_counter_l3);

        ///Constructeur
        cd_counter_l3(sc_module_name name);
};

#endif
```

Fichier cd_counter_l3.cpp :

```
#include "cd_counter_l3.h"

cd_counter_l3::cd_counter_l3(sc_module_name name) : sc_module(name)
{
        //SC_METHOD means veut dire que cette méthode sera exécutée à chaque fois
        //qu'un signal dans la liste de processus est modifié
        SC_METHOD(count_process);
        sensitive_neg << resetx;
        sensitive_pos<<clk;
}


void cd_counter_l3::count_process()
{
        sc_bv<4> count;
```

```
        count = data.read().range(25,22);

        //À l'initialisation, le compte tombe à 0
        if (!resetx.read()){
                countVal = 0;
                end_of_count = true;
        }
        else{
                //Si nous devons enregistrer une nouvelle valeur
                if(setCnt.read()){
                        countVal = sc_uint<4>(count);
                        //Si le compte est à 0, il ne reste qu'une seule et dernière donnée
                        if(sc_uint<4>(count) == 0){;
                                end_of_count = true;
                        }
                        else{
                                end_of_count = false;
                        }
                }
                //Décrémenté, si ce n'est pas la dernière donnée
                else if (decrCnt.read() && countVal.read() !=0){
                        countVal = countVal.read() - 1;
                        if(countVal.read() == 1){
                                end_of_count = true;
                        }
                        else{
                                end_of_count = false;
                        }
                }
                //Sinon, décrémenter et c'est la dernière donnée
                else if (decrCnt.read()){
                        countVal = countVal.read();
                        end_of_count = true;
                }
        }
}
```

Le langage C++ permet de faire de la programmation à un niveau d'abstraction pour lequel il n'est pas toujours possible de générer un circuit matériel qui a un comportement équivalent. Un pointeur, par exemple, trouve difficilement son équivalent en matériel. Un outil qui effectue la synthèse de SystemC doit donc précisément définir quels éléments du langage sont supportés ou ne le sont pas. De façon générale, les outils présentement disponibles commercialement pour effectuer la synthèse de SystemC ne supportent pas les fonctionnalités de la programmation par objets. Néanmoins, un groupe s'est penché sur la possibilité d'utiliser les possibilités qu'offre la programmation par objets pour des fins de synthèse à l'aide d'un outil maison nommé ODETTE [8]. On y retrouve des fonctionnalités telles que l'utilisation de classes dérivées qui réutilisent du code provenant des classes de base et l'utilisation de patrons (*template*) pour définir une fonctionnalité générique.

Puisque traditionnellement ce sont les langages VHDL et Verilog qui permettent de faire la synthèse de circuits numériques, il n'existe qu'un nombre limité de produits permettant de faire la synthèse de SystemC. Trois produits permettent actuellement de faire une telle synthèse : Agility Compiler de Celoxica [2], Cynthesizer de Forte [10] et SystemCrafter SC de SystemCrafter [31]. Jusqu'à tout récemment, SystemC Compiler de Synopsys était également de la partie, mais le produit a été retiré du marché. Présentement, tous ces produits ne supportent pas les mêmes fonctionnalités de la bibliothèque SystemC, ce qui nuit à la portabilité d'un circuit décrit en SystemC. Une description SystemC pour laquelle on peut faire la synthèse avec un outil en particulier risque donc fortement de ne pas fonctionner avec un autre outil de synthèse. C'est pourquoi le OSCI (*Open SystemC Initiative*) travaille présentement à l'élaboration d'une définition du sous-ensemble synthétisable de SystemC. Malheureusement, pour l'instant, la majorité des outils disponibles manquent grandement de maturité.

Un des avantages de SystemC par rapport aux langages de description matérielle traditionnels est que le C++ utilise des fichiers d'entête pour déclarer les composants. Les fichiers d'en-têtes permettent de ne déclarer la description d'un composant qu'à un seul endroit, comparativement aux langages de description matérielle où il est nécessaire de créer des bibliothèques de composants ou de répéter la déclaration d'un composant dans tous les fichiers utilisant le composant. Ceci est un atout très important puisque pour les langages de description matériel, il est beaucoup plus fastidieux d'effectuer une modification à l'interface d'un module. Un deuxième atout de SystemC est que le C++ comprend un précompilateur qui aide grandement à rendre paramétrable la description d'un circuit. Le rôle d'un précompilateur est de faire une première analyse du code pour éliminer ou remplacer certaines sections selon des directives données. Il est donc possible d'intégrer dans une même section de code plusieurs fonctionnalités et de sélectionner au moment de la précompilation les fonctionnalités voulues. Le précompilateur peut par contre rendre la compréhension de sections de code plus difficile

à comprendre et ne permet pas d'instancier plusieurs instances d'un module qui ont des paramètres différents d'une instance à l'autre. L'utilisation du précompilateur dans le cadre de ce projet est approfondie à la section 3.4.1 qui fournit des exemples d'utilisation du précompilateur. La fonctionnalité de patrons (*template*) du C++ pourrait permettre d'instancier différents modules avec chacun des paramètres indépendants, mais cela n'est pas supporté par les outils de synthèse commerciaux. La référence [19] traite en détail l'aspect paramétrable de SystemC au niveau logiciel.

Les langages de description de matériel traditionnels permettent également de rendre paramétrable des circuits, comme le démontre certains projets [23]. En VHDL il est possible d'utiliser les fonctionnalités *generic* pour définir des opérations d'une taille variable et *generate* pour instancier de façon dynamique des composants. La fonctionnalité *generate* se limite par contre à instancier des modules entiers. Il est également possible d'envoyer des valeurs constantes aux entrées d'un composant afin que l'outil de synthèse l'élimine automatiquement. Cette dernière méthode garde le désavantage que certains composants pourraient ne pas être éliminés complètement par l'outil de synthèse dans des circonstances particulières.

1.5 Conclusion

Le présent chapitre a permis de donner au lecteur une vue d'ensemble de la littérature touchant les divers sujets qui seront traités dans ce mémoire de maîtrise. RapidIO, PCI Express, HyperTransport et SPI 4-2 sont tous des protocoles de communication performants utilisés dans l'industrie. Les trois premiers protocoles ont des fonctionnalités et performances similaires. Entre autres, ils utilisent tous plusieurs canaux virtuels pour transmettre l'information et ont plusieurs différents types de requêtes. Ils ont toutefois plusieurs différences, dont le fait que RapidIO et PCI Express soient des protocoles sériels tandis qu'HyperTransport soit un protocole parallèle. SPI 4-2 se différencie des autres protocoles par une plus grande simplicité en ayant qu'un seul

type de paquet. La topologie de base du réseau HT est une chaîne qui comporte un hôte, des tunnels et une cave. HT comporte plusieurs types de paquets, dont ceux de contrôle de flux, les *posted*, *non-posted* et *response*. Les trois derniers types de paquets sont emmagasinés dans des tampons indépendants lorsqu'ils sont reçus.

Bien que la communication interpuces soit un aspect important de bien des systèmes, de plus en plus, il est possible d'intégrer une multitude de composants sur une même puce pour obtenir un système embarqué. Ces systèmes embarqués nécessitent un système pour permettre aux divers composants de communiquer entres eux par l'entremise d'un bus de communication ou d'un réseau embarqué. Les réseaux embarqués permettent d'interconnecter un grand nombre de composants en évitant les problèmes de délai et de complexité de routage que causent les fils de communication entre deux circuits éloignés tout en évitant les limitations de performance des bus. Une grande variété de topologies de réseaux embarqués sont présentement en cours de développement afin de répondre aux besoins futurs des systèmes embarqués. Le réseau RoC utilisé dans cette recherche est un réseau embarqué sur lequel travaille une équipe de recherche de l'École Polytechnique de Montréal.

Il existe plusieurs outils de synthèse qui permettent de créer un circuit à partir de codes sources utilisant la bibliothèque SystemC, mais le plus performant de ces outils a été retiré du marché alors que les outils offerts par les compétiteurs disponibles dans nos laboratoires manquaient de maturité. Ce langage offre des avantages en permettant une grande variété de niveaux d'abstractions. Le précompilateur C++ est un outil idéal pour rendre un circuit paramétrable.

Chapitre 2. Architecture du système

Les systèmes permettant d'effectuer des calculs génériques à l'aide d'un microprocesseur sont très importants à cause de leur aisance à s'adapter à une grande variété d'applications. Avec l'augmentation des besoins de plusieurs applications modernes, il est maintenant souvent nécessaire de distribuer les calculs de tels systèmes sur plusieurs microprocesseurs et à du matériel dédié. Il est donc très intéressant de se pencher sur la création d'une plate-forme de communication qui permet d'intégrer à la fois des microprocesseurs performants, des coprocesseurs et des systèmes embarqués et de leur permettre de communiquer entres eux

2.1 Architecture globale de communication

La communication au sein d'un système de calcul distribué joue un rôle extrêmement important. Une première étape pour faire la conception d'un tel système serait donc d'avoir tous les éléments de communication nécessaires. Du côté de la communication intrapuce, il existe déjà plusieurs groupes de recherche qui se penchent sur le développement de plates-formes de systèmes embarqués utilisant des bus de communication et des réseaux embarqués. Il est donc peu pertinent dans le cadre de ce projet d'explorer cette question. Par contre, du côté interpuces, il existe plusieurs technologies, mais l'accès à ces technologies par l'achat de circuits préconçus peut être très coûteux. Il est donc très intéressant d'explorer la conception d'un lien de communication interpuces pour mieux comprendre cette technologie.

Puisque plusieurs types de liens de communication interpuces à grande performance ont une portée physique limitée, il faut prévoir qu'un système de grande taille devra utiliser d'autres types de liens de communication à grande portée comme Infiniband ou Ethernet. Bien qu'il soit pertinent de prévoir que des liens à grande portée

pourraient potentiellement être intégrés à un système, cette fonctionnalité sort du cadre de ce projet.

2.2 Choix d'une technologie de lien interpuces

La première étape de la conception d'un système permettant une communication entre puces est d'établir le protocole de communication à utiliser. Puisqu'il existe déjà plusieurs technologies librement disponibles, il n'est que logique d'utiliser une de celles-ci, puisque leur bon fonctionnement est prouvé et cela permet aussi une plus grande interopérabilité avec d'autres produits. Le choix peut donc se faire non seulement sur les aspects techniques du protocole, mais également sur la base de son succès commercial. Il est souvent avantageux d'opter pour une technologie qui est flexible, ce qui permet de l'utiliser pour plusieurs aspects d'un système. Cela permet de concentrer le développement sur un seul protocole plutôt que sur plusieurs et d'avoir à faire la conception de ponts pour passer d'un protocole à un autre. Puisque nous voulons faire la conception d'un système de calcul générique qui est facilement extensible, le support de fonds de panier est important. Et puisque nous voulons communiquer avec des microprocesseurs, c'est un atout si le protocole de communication est approprié pour communiquer directement avec ceux-ci.

Vu que SPI est surtout utilisé pour faire la transmission de flux de données en continu, il est moins approprié pour être utilisé comme bus de microprocesseur et il n'est donc pas retenu. PCI Express n'est également pas retenu puisque bien qu'il semble bien répondre à nos besoins, sa spécification n'est pas librement disponible sans être un membre du groupe PCI-SIG et, au moment de débuter le projet, aucun produit PCI Express n'était commercialisé. RapidIO et HyperTransport sont donc les deux technologies de choix pour répondre à nos besoins. Avant de se pencher sur quelle technologie devait être utilisée, HT avait déjà été sérieusement étudiée et il était donc avantageux de procéder avec cette technologie.

La topologie de base de HT est une chaîne et cela est très approprié pour lier quelques composants. Par contre, pour un système plus extensible, il est nécessaire d'envisager l'utilisation d'une topologie en étoile qui requière des commutateurs. HT ayant une spécification assez complexe, il a été jugé plus prudent de ne pas se lancer directement dans la conception d'un commutateur mais plutôt d'opter pour un tunnel qui est un élément HT à deux liens. Un tunnel est un excellent compromis puisqu'il peut être utilisé à la fois comme tunnel ou comme une cave HT à l'intérieur d'une chaîne. En plus, le fait qu'un tunnel comporte plusieurs liens fait en sorte qu'il est probable qu'une grande quantité de ses sous-modules pourraient être réutilisés pour faire la conception d'un commutateur. À cause de la complexité de HT, il était également prudent de commencer par valider l'architecture d'un tunnel et de son fonctionnement par une spécification exécutable. SystemC est un élément idéal pour concevoir une spécification exécutable d'un circuit.

2.3 Interaction avec un système embarqué

Permettre une interaction transparente entre un protocole de communication interpuces et un système embarqué permettrait à une multitude de composants distribués sur plusieurs systèmes embarqués de communiquer entre eux de façon transparente pour partager efficacement des tâches de grande complexité. La figure 2.1 montre comment plusieurs réseaux embarqués (RE) pourraient interagir à l'intérieur d'un système en utilisant HT.

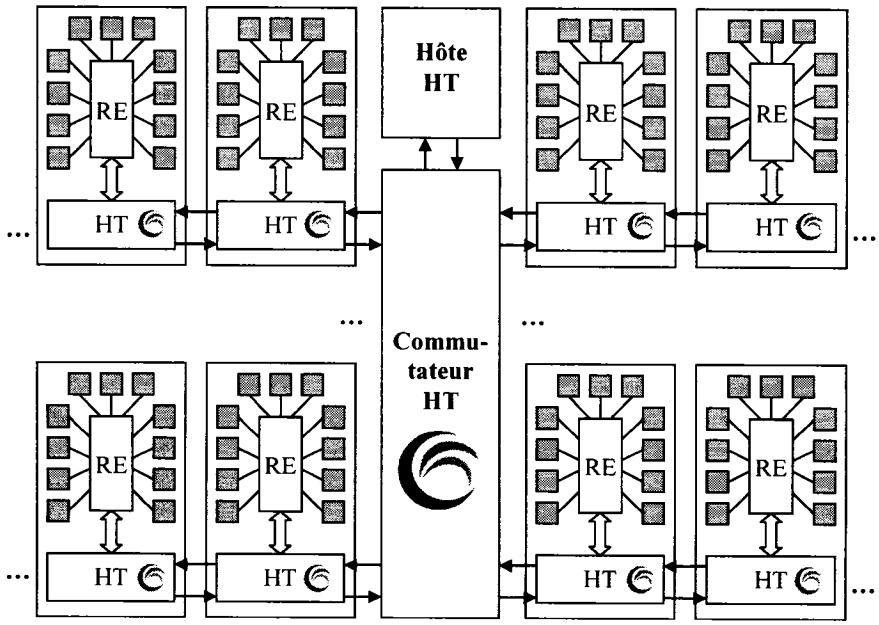

Figure 2.1 - Intégration HT avec des réseaux embarqués (RE)

La communication à l'intérieur des systèmes embarqués peut prendre plusieurs formes, mais les opérations permises sont en général relativement simples afin de limiter la complexité logique de son implémentation. Pour interagir avec des systèmes embarqués, il faut donc choisir une série d'opérations qui sont supportées et qui pourront facilement transiter par le protocole HT. Trois transactions courantes sont les écritures, les lectures et les réponses aux lectures, trois opérations qui sont également supportées par HT. Il est donc relativement facile d'encapsuler une requête provenant d'un système embarqué pour la distribuer par un réseau HT. Cette tâche nécessite un pont qui s'occupe d'encapsuler les paquets provenant d'un système embarqué pour les faire passer sur un réseau HT et également de décapsuler les paquets HT pour les transmettre au système embarqué. La conception d'un pont pour passer d'un réseau embarqué à un tissu de communication HT est donc un aspect important du projet.

2.4 Conclusion

Les systèmes qui effectuent des calculs génériques peuvent bénéficier de flexibilité puisque cela leur permet de répondre aux besoins d'une grande diversité d'applications. Cette flexibilité peut être obtenue en permettant à un grand nombre d'unités de calcul de communiquer entres elles à l'aide d'un protocole de communication interpuces. HT est une technologie qui répond parfaitement aux besoins de communication d'un tel système, mais l'accès à cette technologie est difficile à cause du coût élevé des circuits commerciaux pour le contrôle de communication HT. Le mémoire se penche donc sur la conception d'un tunnel HT afin d'obtenir accès à cette technologie. Puisque l'on retrouve de plus en plus de développement de systèmes embarqués, il est important que l'architecture de communication interpuces permette la communication avec ceux-ci. Un pont doit être conçu afin de faire passer de l'information d'un système embarqué à un réseau de communication interpuces HT.

Chapitre 3. Architecture du tunnel HyperTransport

Le protocole HT a été sélectionné comme tissu de communication interpuces dans notre système. Un tissu de communication complexe requiert des commutateurs, mais avant de se lancer dans la conception d'un design aussi complexe, il est préférable d'avoir une bonne compréhension du problème et de sa complexité. C'est pourquoi un premier objectif est de faire la conception d'un tunnel HT, ce qui permettra également d'obtenir une bonne compréhension de la complexité de conception et de la quantité de ressources matérielles que requiert un tel design. Le présent chapitre commence donc par donner un aperçu de l'architecture du tunnel HT pour ensuite aborder la méthode de conception, la licence de développement du tunnel, les aspects paramétrables du tunnel et les détails des composants individuels de l'architecture et finalement une comparaison à d'autres architectures commerciales actuelles. Le prochain chapitre poursuivra avec la planification de la vérification du design ainsi qu'avec la présentation des résultats obtenus.

3.1 Licence de développement et de distribution

Un des buts de cette recherche est d'obtenir une meilleure compréhension d'un tunnel HT et de partager nos résultats avec la communauté scientifique. La publication d'articles demeure un outil privilégié pour partager nos résultats, mais il est également possible de partager notre travail de façon plus directe. La conception de circuits numériques se fait à l'aide de l'écriture de codes et il est possible de partager ce code à l'aide d'une licence de type libre. Une licence libre permet à d'autres gens d'utiliser le code tant qu'ils mentionnent clairement l'utilisation de notre travail. Nous avons donc décidé de publier le tunnel HT sur le site web opencores.org sous une licence Mozilla Public Licence (MPL).

3.1.1 Choix de licence

Il existe deux grandes catégories de licences ouvertes : celles dites « copyleft » et celles dites « non copyleft ». Avec une licence « copyleft », un utilisateur d'un code couvert par ce type de licence doit distribuer le code avec toute modification y ayant été apportée. Dans le cas d'une licence « non copyleft », un utilisateur n'a pas à distribuer le code, il n'a qu'à mentionner son utilisation du code. Une licence de type « copyleft » est donc plus contraignante pour l'utilisateur, mais elle s'assure que le produit continue à évoluer au fur et à mesure que des modifications y sont apportées. La licence MPL utilisée pour le tunnel est de type « copyleft ».

Il existe plusieurs licences « copyleft », dont la très populaire General Public Licence (GPL) ainsi que la Lesser General Public Licence (LGPL). Les licences GPL et LGPL ne sont malheureusement que ciblées pour du logiciel et ne couvrent pas adéquatement un code utilisé pour produire un circuit avec un outil de synthèse. La licence MPL, quant à elle, couvre tout ce qui est dérivé du code, donc couvre adéquatement un circuit généré à partir de code source. La licence MPL offre également facilement la possibilité d'avoir un code source couvert sous deux licences, au choix de l'utilisateur. Le tunnel exploite cette possibilité et est donc couvert par deux licences : MPL et une licence donnant des droits exclusifs à l'École Polytechnique de Montréal. Si jamais nous voulions faire une utilisation commerciale du tunnel sans être limité par la licence MPL, il serait donc possible de le faire.

3.1.2 Distribution

Bien que le code source soit couvert par une licence ouverte, il faut un moyen pour distribuer celui-ci. Une façon simple de procéder serait de placer le code source sur un serveur de l'École Polytechnique, mais nos serveurs n'ont pas nécessairement toute l'infrastructure pour supporter une communauté de développeurs qui travaillent à l'amélioration d'un design tout en permettant un accès pour télécharger ce code

facilement à partir d'un site web. C'est par contre exactement ce qu'offre le site web opencores.org, soit une grande variété de projets de circuit libres.

Opencores.org utilise un système CVS (Concurrent Versions System) qui permet de garder un historique de toutes les modifications apportées au code source, ainsi qu'un éditeur de site web avec une interface web pour permettre de maintenir la page web du projet. Tous les projets sur opencores.org respectent certaines règles [24] pour les rendre plus uniformes auprès des utilisateurs.

3.2 Aperçu de l'architecture

Le tunnel est conçu pour être compatible à la spécification HyperTransport 2.00b ainsi que pour pouvoir être utilisé sous une variété de plates-formes incluant les FPGAs et la technologie de cellules standards. Pour demeurer générique, le design n'intègre pas de mémoires embarquées, mais utilise plutôt des ports pour accéder à des mémoires embarquées externes.

3.2.1 Options supportées

HyperTransport offre une grande variété d'options et chaque design doit choisir lesquelles sont supportées ou non. Voici une liste de fonctionnalités qui sont supportés par le tunnel :

- Mode *retry*
- DirectRoute
- Lien de 2, 4 ou 8 bits
- Réorganisation de paquets à l'intérieur des canaux virtuel

Par défaut, HT ne fait pas de correction d'erreur mais plutôt tout simplement une vérification périodique pour détecter si une erreur est survenue. Plus la fréquence d'opération des liens HT est élevée, plus il devient probable qu'une erreur se glisse dans

la communication et c'est pourquoi un mode de correction d'erreurs nommé « mode *retry* » a été introduit dans la spécification. Ce mode permet d'insérer un code CRC (contrôle par redondance cyclique) à la fin de chaque paquet pour vérifier son intégrité et permettre la retransmission du paquet lorsqu'une erreur survient. Dans ce mode, il est donc nécessaire de garder un historique de tous les paquets ayant été envoyés, et cela jusqu'à ce que leur réception soit confirmée.

Un paquet envoyé par un élément d'une chaîne doit normalement toujours être envoyé vers l'hôte de la chaîne qui, lui, va le réfléchir vers la destination finale si nécessaire. Cela a l'avantage de garantir le modèle producteur/consommateur du protocole PCI, mais a par contre le désavantage de causer une latence importante et d'augmenter la bande passante nécessaire puisque les paquets sont présents dans la chaîne pour une plus grande durée. Le mode DirectRoute permet à deux noeuds d'une chaîne de communiquer directement entres eux avec un modèle producteur/consommateur restreint.

Pour supporter adéquatement le modèle producteur/consommateur, HT place certaines limitations sur quels paquets peuvent passer avant d'autres paquets. En temps normal, les paquets des canaux virtuels *non-posted* et *response* ne peuvent pas passer les paquets du canal virtuel *posted.* Lorsque le modèle producteur/consommateur n'est pas requis, il est avantageux que les paquets ne soient pas bloqués par d'autres canaux virtuels et le champ *passPW* des paquets HT permet cela. Pour éviter qu'un paquet ayant le champ *passPW* activé reste bloqué derrière un paquet qui n'a pas ce champ activé, le tunnel permet que les paquets avec *passPW* activé puissent les dépasser.

3.2.2 Aperçu des modules

Le tunnel HT se divise en une série de modules tels que présentés à la figure 3.1, qui présente les chemins des données principaux entre ces modules. Plusieurs chemins de données peu critiques et chemins de contrôle sont négligés dans la figure 3.1. Le

tunnel est divisé en modules afin de faciliter la conception et la vérification. Puisqu'un tunnel comporte deux liens de communication identiques, une grande majorité des modules sont présents à deux reprises à l'intérieur du tunnel.

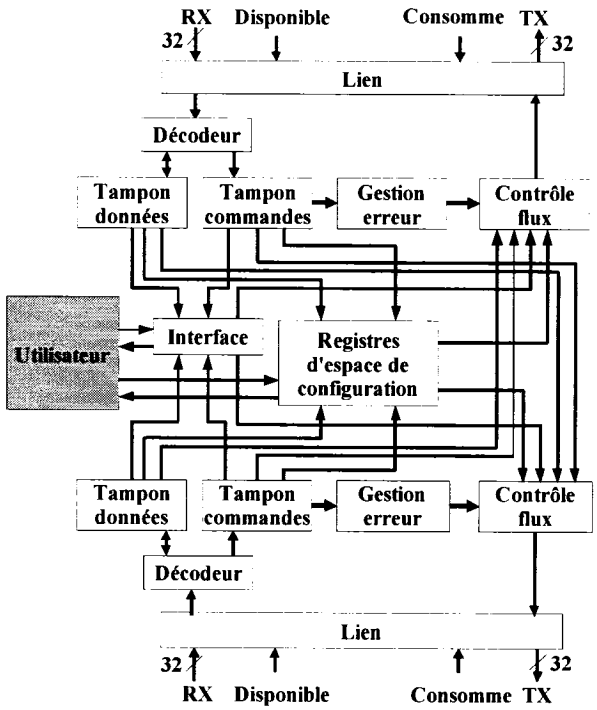

Figure 3.1 - Architecture du tunnel HyperTransport

Le *Lien* permet d'initialiser la communication et de gérer la vérification périodique d'erreur. Ce module permet donc de cacher au reste du design plusieurs détails de la communication. Le *Décodeur* analyse les données reçues pour en faire des paquets qui sont ensuite enregistrés dans des tampons. Les données sont emmagasinées dans le module *Tampon de données* et les paquets de commande sont emmagasinés dans le module *Tampon de commandes*. Le module *Tampon de données* gère l'enregistrement et l'accès aux données, tandis que le module *Tampon de commandes* fait un travail équivalent pour les paquets de commande, en plus de faire la réorganisation des paquets et de les envoyer à la destination appropriée. Lorsque des paquets sont reçus par erreur, le module *Gestion erreur* s'occupe de traiter et de répondre à ces paquets si nécessaire.

Le module *Contrôle de flux* gère les paquets qui doivent être envoyés et fait également la gestion de flux pour éviter d'envoyer plus de paquets que la destination ne peut en traiter.

Le module *Registres d'espace de configuration* (*REC*) est un espace mémoire que peut accéder l'hôte de la chaîne pour vérifier les fonctionnalités supportées par le tunnel et pour configurer ces fonctionnalités, par exemple pour définir quelle plage d'adresse du système est allouée au tunnel. L'*Interface* permet à un utilisateur (circuit utilisant le lien HT) d'interagir avec le tunnel facilement pour envoyer et recevoir des paquets sans avoir à se soucier du fait qu'un tunnel a deux liens physiques.

3.3 Méthode de conception

La conception d'un circuit logique est une tâche souvent complexe et c'est pourquoi il est nécessaire de commencer par obtenir une spécification et ensuite de bien définir l'architecture du système en le séparant en modules et en établissant les interfaces entre ceux-ci. Dans le cas de HT, la spécification du protocole existe déjà, donc la première étape consiste à définir l'architecture et les interfaces entres les modules. À la suite de cette étape, plusieurs options de flots de conception sont disponibles telles qu'elles sont démontrées à la figure 3.2.

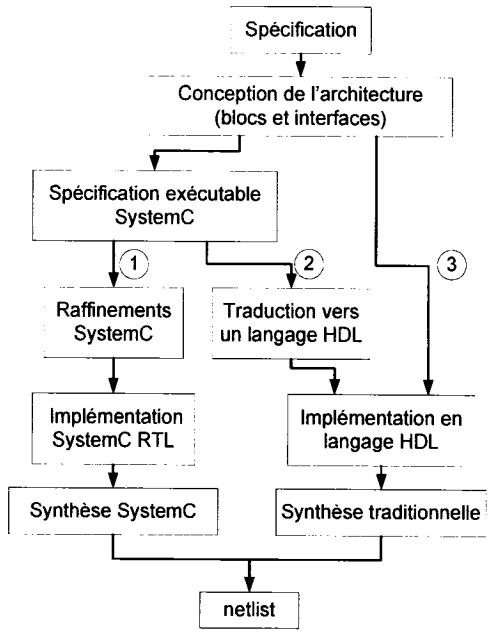

Figure 3.2 - Divers flots de conception avec SystemC

Traditionnellement, une fois les blocs et interfaces bien définis, il est possible de passer directement à la conception du matériel à l'aide d'un langage de description de matériel (HDL) tel qu'on peut le voir avec le flot de conception 3 dans la figure 3.2. Mais il est souvent préférable de prouver le bon fonctionnement de l'architecture d'un design complexe avant de se lancer dans sa conception matérielle. Une façon d'effectuer cette vérification est de créer une spécification exécutable à l'aide de SystemC. Une fois la spécification exécutable terminée, habituellement il faut traduire la spécification exécutable vers un langage de description de matériel pour poursuivre un flot de conception standard : c'est le flot 2. Le flot numéro 1 est une autre option possible si la spécification exécutable a été créée à un niveau d'abstraction suffisamment bas : il est possible de raffiner la spécification exécutable pour obtenir une description SystemC synthétisable.

Pour le tunnel HT, c'est le flot de conception numéro 1 qui a été adopté. Après avoir bien défini un contrat d'interfaces entre les divers modules, les différents modules ont été conçus de façon logicielle en utilisant la bibliothèque SystemC. Bien que la

spécification exécutable originale ne supportait pas le mode *retry* et comportait plusieurs lacunes, elle a permis de vérifier le bon fonctionnement de l'architecture. Une fois que la spécification exécutable a démontré que l'architecture du système était fonctionnelle, l'outil SystemC Compiler de Synopsys a permis de faire la synthèse de parties existantes de la spécification exécutable. Au lieu de tout devoir implémenter dans un langage de description de matériel, il était donc avantageux de modifier la spécification exécutable pour la rendre entièrement synthétisable.

En même temps que la spécification exécutable a été modifiée afin qu'elle soit synthétisable, le mode *retry* ainsi que d'autres améliorations diverses ont été ajoutés au design. Des bancs d'essai ont été conçus pour la grande majorité des niveaux hiérarchiques de l'architecture. Finalement, un nettoyage du code et des optimisations ont été apportées pour améliorer la performance, rendre le tunnel plus paramétrable et diminuer sa complexité.

3.4 Aspects paramétrables

Un but premier du projet de recherche est d'obtenir un circuit qui soit le plus paramétrable possible afin de s'adapter à un maximum d'applications. Certains paramètres permettent tout simplement de faire la configuration du fonctionnement du tunnel, tandis que d'autres peuvent permettre d'éliminer des fonctionnalités complètes afin de réduire la complexité du circuit. La majorité des paramètres utilisent le précompilateur, tandis que d'autres utilisent des constantes globales. Les options paramétrables dans le tunnel sont les suivantes :

- Support du mode *retry* (optionnel);
- Support de la fonctionnalité DirectRoute (optionnel);
- Réorganisation des paquets à l'intérieur des canaux virtuels (optionnel);
- Alignement des données : effectué par le lien ou par le désérialiseur externe;
- Latence de l'interface pour communiquer avec l'utilisateur;

- Lien configurable à une largeur de 2, 4 ou 8 bits;
- Profondeur des tampons de données;
- Profondeur des tampons de commandes;
- Nombre et taille des registres BAR (Base Address Registers – ils définissent la plage de mémoire utilisée par le tunnel).

3.4.1 Exemples d'utilisation du précompilateur

Le précompilateur C++ peut être utilisé pour rendre paramétrable une très grande variété de fonctions d'un circuit. Cette section présente différentes façons dont le précompilateur peut être utilisé à cet effet. L'option d'utiliser un alignement interne des données pour la réception de données est utilisée dans le cadre de ces exemples.

- Pour modifier les signaux d'entrée et de sortie du module. Dans cet exemple, les sorties `lk_deser_stall_phy` et `lk_deser_stall_cycles_phy` ne sont présentes que lorsque l'alignement interne des données n'est pas activé :

```
#ifndef INTERNAL_SHIFTER_ALIGNMENT
sc_out<bool > lk_deser_stall_phy;
sc_out<sc_uint<LOG2_CAD_IN_DEPTH> > lk_deser_stall_cycles_phy;
#endif
```

- Pour modifier la liste de sensibilité d'un processus. Dans cet exemple, le processus `detect_ctl_transition_error` n'est sensible aux signaux `delayed_reordered_ctl` et `frame_shift_div_width` que si l'alignement interne des données est activé :

```
  SC_METHOD(detect_ctl_transition_error);
  sensitive << state << reordered_data_ready << reordered_ctl
#ifdef INTERNAL_SHIFTER_ALIGNMENT
         << delayed_reordered_ctl << frame_shift_div_width
#endif;
```

- Pour qu'un processus soit entièrement éliminé. Dans cet exemple, le processus `output_reordered_cad_and_ctl` n'est présent que si l'alignement interne des données n'est pas activé.

```
#ifndef INTERNAL_SHIFTER_ALIGNMENT
```

```
  SC_METHOD(output_reordered_cad_and_ctl);
  sensitive << reordered_cad << reordered_ctl;
#endif
```

- Pour modifier le comportement interne d'un processus. Dans cet exemple plus complexe, on peut voir une expression de type « if » qui contient des paramètres qui changent si l'alignement interne de données est activé et si le mode *retry* est implémenté :

```
#ifdef INTERNAL_SHIFTER_ALIGNMENT
   if(   reordered_data_ready.read() &&
#else
   if( reordered_data_ready_tmp &&
#endif
#ifdef RETRY_MODE_ENABLED
      !retry_disconnect && !new_detected_ctl_transition_error.read() &&
#endif
      !ldtstop_disconnect_rx.read()){
   framed_data_available = true;
}
```

- Pour modifier le nombre d'états d'une machine à états. Ici il est possible de voir que l'état `RX_FRAME_RETRY_DISCONNECT_ST` n'est disponible que si le mode *retry* est implémenté.

```
enum rx_frame_state {
  RX_FRAME_INACTIVE_ST,
  RX_FRAME_WAIT_FRAME_ST,
  RX_FRAME_ACTIVE_ST,
#ifdef RETRY_MODE_ENABLED
  RX_FRAME_RETRY_DISCONNECT_ST,
#endif
  RX_FRAME_LDTSTOP_DISCONNECT_ST
};
```

3.5 Architecture détaillée

Nous avons présenté précédemment un bref aperçu de la fonctionnalité des divers modules du tunnel. La présente section approfondit le sujet en donnant des détails sur chacun des modules et également des sous modules.

3.5.1 Domaines d'horloge

La fréquence d'opération des liens physiques de HT est très élevée afin d'obtenir une grande bande passante. Cette fréquence des liens physiques est en général beaucoup

plus élevée que la fréquence maximale d'un circuit numérique relativement complexe. C'est pourquoi le tunnel HT doit traiter l'information provenant des liens en parallèle, de façon à pouvoir opérer à une fréquence d'opération beaucoup plus basse que la fréquence des liens.

Puisque tous les paquets HT sont des multiples de 32 bits, le tunnel traite également les données par tranches de ce même multiple. Il est donc nécessaire de désérialiser les données entrantes et de sérialiser les données sortantes. Dans le cas d'un lien physique de 2 bits, le tunnel utilise un facteur de sérialisation de 16 et fait la lecture et l'écriture de 2 x16 bits par coup d'horloge. Dans le cas d'un lien de 4 bits et 8 bits, on retrouve des facteurs de sérialisation de 8 et 4 respectivement. La largeur du lien physique est configurable lors de la synthèse du design. Bien que le choix de traiter les données 32 bits par coup d'horloge semble logique, nous allons voir plus tard que ce n'était pas nécessairement le bon choix à faire, puisqu'il aurait été possible d'augmenter de façon significative la bande passante du tunnel en traitant 64 bits ou même plus par coup d'horloge.

Un des aspects très important des protocoles de communication interpuces est que les puces interreliées par ces protocoles ne peuvent souvent pas toutes être dans le même domaine d'horloge. Si deux éléments d'une chaîne HT partagent la même base de temps pour déterminer l'horloge de leur système, bien que les deux éléments ne soient pas nécessairement en phase, leur fréquence d'opération sera identique : c'est le mode synchrone. Dans ce cas, les deux éléments HT produisent et consomment des paquets à la même fréquence. Mais dans le cas où les deux éléments HT ne partagent pas la même base de temps, il est possible que les deux éléments produisent et consomment des données à des fréquences différentes : c'est le mode asynchrone. Le tunnel supporte les deux modes d'opération.

La spécification HT permet à un élément HT de dévier de la fréquence d'opération spécifiée de 1000 ppm (partie par million), donc pour une différence maximale de 2000 ppm entre deux éléments d'une chaîne HT. La spécification HT propose une méthode de compensation des différences de cadence d'horloge fondée sur le fait qu'une partie de la logique fonctionne sur la base de temps de l'horloge de réception de données (l'horloge de transmission du noeud précédent). Étant donné que, pour chaque 512 octets transmis, un code CRC de 4 octets est inséré dans le flux de données, il est possible d'éliminer du flux de données ces 4 octets après avoir vérifié l'exactitude du code, afin de compenser pour un transmetteur qui a une fréquence d'opération légèrement supérieure à celle du récepteur.

Une méthode autre que celle proposée par la norme HT a été adoptée; cette méthode, qui n'affecte pas la compatibilité avec la norme, consiste à faire en sorte que le tunnel ait une fréquence d'opération supérieure à la fréquence maximale d'opération des autres éléments HT. On se retrouve donc avec une séparation de domaines d'horloge telle qu'on peut le voir à la figure 3.3.

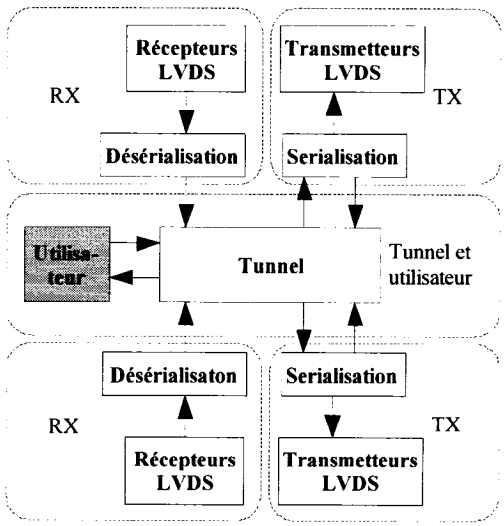

Figure 3.3 - Domaines d'horloge du tunnel

Il est important de noter que le tunnel est entièrement contenu dans un domaine d'horloge et que seuls les couches physiques de sérialisation et de désérialisation ainsi que les récepteurs et transmetteurs LVDS sont dans des domaines d'horloge différents. Les deux domaines d'horloge RX fonctionnent à la fréquence d'opération de l'horloge de réception de donnée fournie par les autres éléments HT, les domaines d'horloge TX fonctionnent à la fréquence d'opération du lien HT ± 1000ppm et doivent rester en phase avec le domaine RX et finalement le tunnel fonctionne à une fréquence supérieure à la fréquence du lien HT + 1000ppm. La fréquence d'opération du tunnel est donc supérieure à la fréquence maximale du lien de réception et il n'est donc pas possible que les données arrivent plus vite que la vitesse à laquelle le tunnel peut les traiter. Le tableau suivant donne les avantages et les désavantages de cette méthode par rapport à la méthode de compensation d'horloge.

Tableau 5 - Méthodes pour pallier aux différences d'horloges entre deux éléments HT

	Compensation à l'aide du CRC périodique	**Fréquence d'opération du tunnel supérieure à la fréquence maximale d'opération du lien**
Avantages	- Ne nécessite que la génération d'une seule horloge de référence pour le tunnel à Freq ± 1000ppm	- La partie logique du tunnel est entièrement comprise à l'intérieur d'un seul domaine d'horloge
Désavantages	- Un partie de la logique du tunnel est dans un domaine d'horloge différent : il est nécessaire de faire passer plusieurs signaux entres les domaines d'horloges - Nécessite un tampon *premier entré premier sorti* pour effectuer la compensation, ce qui peut augmenter la latence du tunnel.	- Nécessite la génération de deux horloges, une à Freq ± 1000 ppm comme référence pour les domaines TX et une autre à une fréquence plus élevée que Freq + 1000 ppm pour la logique du tunnel.

3.5.2 Initialisation du circuit

Une grande majorité des circuits synchrones doivent être initialisés lors de leur démarrage pour s'assurer qu'ils sont dans un état connu. Cela est normalement effectué à l'aide d'un signal de réinitialisation qui est distribué à tout le circuit. HT est un cas assez particulier, puisqu'il comporte deux signaux d'initialisation : une initialisation dite « à froid » et une autre dite « à chaud ». L'initialisation « à froid » n'est effectuée qu'une seule fois lors du démarrage du système et elle s'applique au tunnel entier. L'initialisation « à chaud » permet de réinitialiser la communication dans le tunnel tout en gardant une certaine quantité d'informations enregistrées dans les registres d'espace de configuration (le module *REC*), tel que la plage de mémoire qu'utilise le tunnel. Certains changements de paramètres, entre autres l'activation du mode *retry*, ne prennent effet qu'après une séquence d'initialisation « à chaud ».

Bien que cela soit inhabituel, à cause des requis de la spécification de HyperTransport, le tunnel HT comporte deux signaux d'initialisation de type asynchrone. Il est par contre important de noter que très peu de bascules sont sensibles à la réinitialisation de type « à froid » et il serait donc facilement possible de faire une initialisation de type synchrone pour ces bascules si cela était nécessaire.

3.5.3 Lien

Le rôle premier du module *Lien* est d'initialiser la communication ainsi que de calculer et de vérifier le CRC périodique. Le *Lien* est également responsable d'initialiser la communication après l'initialisation du circuit, après une séquence LDTSTOP qui permet d'économiser de l'énergie et finalement après une séquence *retry* qui permet de corriger une erreur de transmission. Le lien se sépare en trois sous-modules : RX, TX et vérification et calcul de CRC tel qu'il est montré à la figure 3.4. On y voit les grandes lignes du flot des données à l'intérieur du module. La partie vérification et calcul de

CRC comporte deux machines à états finis (MÉF). On y retrouve en plus une quantité de connexions pour contrôler l'état de la connexion.

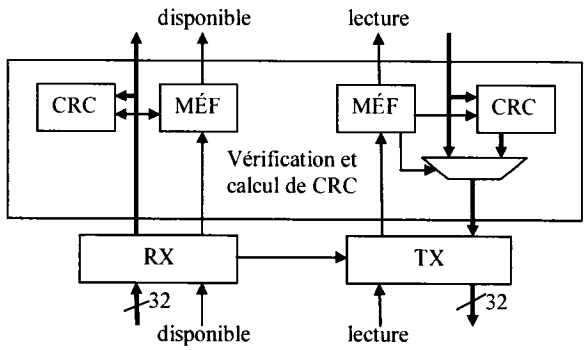

Figure 3.4 - Schéma bloc du *Lien*

Le module TX a le rôle relativement simple de générer une séquence d'initialisation lorsque nécessaire. Lorsque le lien est initialisé, il laisse alors passer les données provenant du module de vérification et de calcul de CRC. Le module TX a également le rôle d'effectuer une sérialisation des données lorsque la largeur du lien est plus petite que la largeur naturelle du lien. Une largeur du lien plus petite que sa largeur naturelle peut survenir lorsqu'un tunnel synthétisé pour une certaine taille de lien communique avec un autre élément HT qui a une taille de lien plus petite.

Le module de vérification et de calcul de CRC, quant à lui, insère périodiquement des codes CRC dans le flot de données sortant et retire les codes CRC du flot de données entrant. La figure 3.4 montre que ce même sous-module ne modifie pas les données entrantes: Lorsque les codes CRC sont reçus, le module de vérification et de calcul de CRC indique tout simplement qu'il n'y a pas de données disponibles.

Le module RX a comme rôle premier de détecter la séquence d'initialisation pour être en mesure de bien déterminer le début des données. Les grandes lignes de ce sous-module sont présentées à la figure 3.5. Comme le module TX, le module RX doit dans un premier temps traiter le cas où le lien est plus petit que la largeur naturelle du lien, et

cela en effectuant une désérialisation supplémentaire des données entrantes si nécessaire. Un deuxième aspect que le module RX doit traiter est la possibilité que les données entrantes ne soit pas alignées correctement par rapport à l'alignement naturel des paquets de HT. Certains désérialiseurs externes permettent de corriger un alignement incorrect mais, si cela n'est pas le cas, un décaleur doit être ajouté (optionnel) pour corriger l'alignement. Lorsque l'initialisation est amorcée et que l'élément HT avec lequel on communique est prêt à recevoir des données, le module RX avertit le module TX qu'il peut terminer sa séquence d'initialisation.

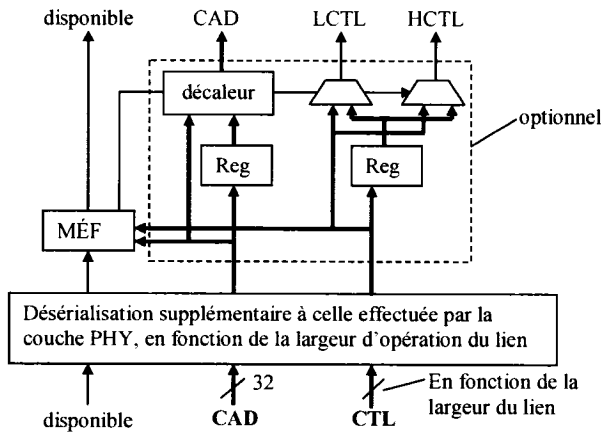

Figure 3.5 - Schéma bloc du sous-module RX du *Lien*

3.5.4 Décodeur

Une fois que les données sont reçues, alignées et que les codes CRC périodiques sont retirés, le *Décodeur* analyse celles-ci pour en faire des paquets. L'en-tête des paquets est en premier lieu décodé, ce qui permet de déterminer le type et la taille des paquets de commande. Les paquets de type NOP sont directement envoyés au module *Contrôle de flux*, puisqu'ils contiennent des données sur les crédits de tampons et sur la bonne réception des paquets lorsque le mode *retry* est activé. Les autres types de paquets de commande sont enregistrés dans des registres avant d'être passés aux tampons de commande. La figure 3.6 qui suit montre l'architecture générale du décodeur.

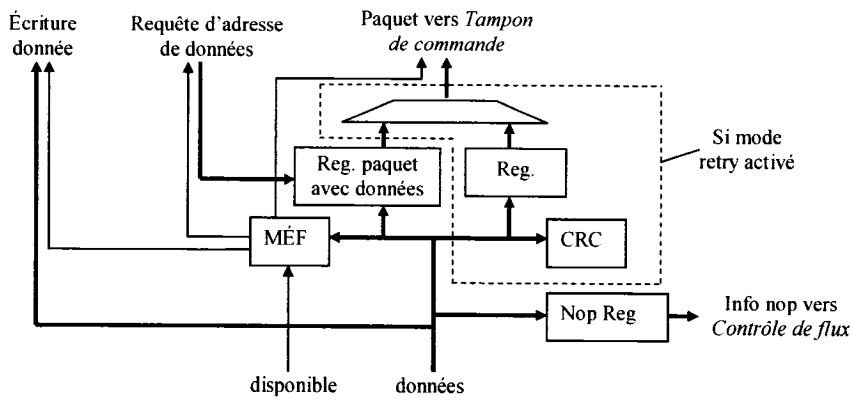

Figure 3.6 - Schéma bloc du *Décodeur*

Lorsque le mode *retry* est activé, le *Décodeur* a deux registres de paquets : un pour les paquets qui ont des données qui suivent et un autre pour les paquets sans données. Les paquets doivent être retenus jusqu'à ce qu'ils soient validés avec leur code CRC. Puisque les paquets avec données et ceux sans données peuvent être entrelacés, il est donc nécessaire d'avoir deux registres. Il est également nécessaire d'avoir un module de calcul de CRC qui peut mémoriser deux valeurs de CRC : un pour les paquets avec des données associées et l'autre pour les paquets qui n'ont pas de données associées.

Lorsque le mode *retry* n'est pas activé, il n'est pas nécessaire d'avoir deux registres, puisqu'il n'est pas requis d'attendre que toutes les données soient arrivées avant d'envoyer vers les tampons de commandes un paquet qui a des données qui suivent. *Note : Les données de synthèse ont été obtenues pour un décodeur qui a deux registres de paquets, peu importe si le mode retry est activé ou non.* Un registre pour retenir un paquet est toutefois encore nécessaire, puisque HT requiert qu'un paquet ne soit pas validé tant qu'un paquet de commande suivant n'est pas reçu, pour éviter que de la corruption ne survienne si une réinitialisation du lien avait débuté après le début de la transmission du paquet.

Lorsqu'un paquet a des données associées à celui-ci, une adresse où enregistrer les données est demandée au *Tampon de données*. Cette adresse est enregistrée avec le

paquet et celle-ci servira à récupérer les données par la suite. Les données qui suivent sont alors envoyées directement au *Tampon de données*. Dans le cas où un paquet avec une adresse de 64 bits est reçu, les bits d'adresse supplémentaires ne sont pas sauvegardés et un bit d'erreur est activé pour ce paquet. Ce bit d'erreur permettra ensuite de diriger le paquet vers le gestionnaire d'erreur qui pourra générer une réponse appropriée à ce paquet si nécessaire.

3.5.5 Tampon de données

Puisque seuls les paquets de commande contiennent l'information nécessaire pour déterminer le rôle et la destination des paquets de données, il est préférable de mettre les données de côté dans des tampons jusqu'à ce que l'usage de celles-ci soit requis. Le *Tampon de données* permet de répondre à cette tâche en emmagasinant les paquets de données reçus par le lien de communication jusqu'à ce qu'ils soient récupérés ou effacés ultérieurement. Puisqu'il est nécessaire d'emmagasiner une quantité considérable de données, il est préférable d'utiliser des mémoires embarquées plutôt que des bascules.

Même si des mémoires embarquées sont utilisées, il est nécessaire de savoir quels tampons sont disponibles à l'aide de bascules. Chaque position dans les tampons contient une bascule telle qu'elle est montrée à la figure 3.7. Lorsqu'un nouveau paquet est enregistré à une position, une bascule enregistre que cette position n'est plus libre et lorsque le paquet est effacé, cette même bascule enregistre que la position est maintenant disponible.

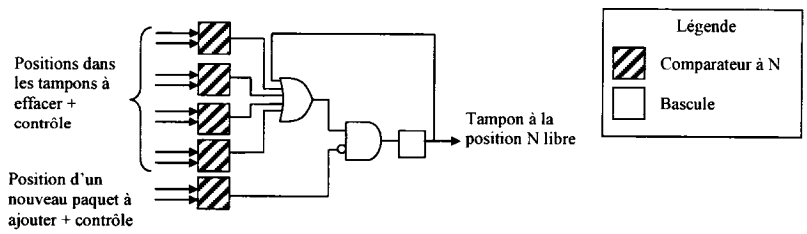

Figure 3.7 - Bascule pour mémoriser si une position est disponible dans le *Tampon de données*

Ce circuit de contrôle de tampon est répété pour chaque position disponible dans les tampons et la figure 3.8 montre comment ils s'intègrent dans le module. Ces circuits de contrôle sont répartis entre les trois grandes catégories de paquets, soit *posted*, *non-posted* et *response*. Pour chacune des catégories, un encodeur de priorité détermine la première position disponible et un multiplexeur vient sélectionner la bonne adresse en fonction du type de données à enregistrer.

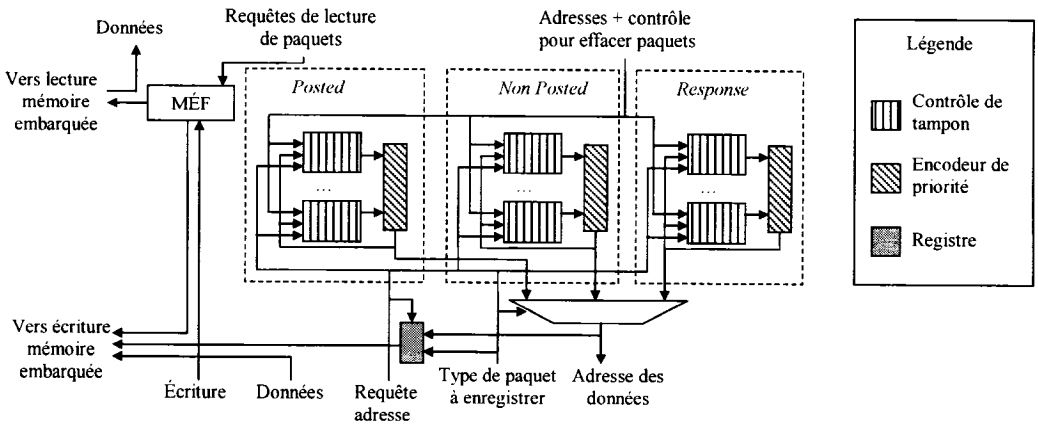

Figure 3.8 - Schéma bloc du *Tampon de données*

Une machine à états contrôle l'écriture et la lecture de paquets tout en effectuant la gestion des crédits de contrôle de flux pour les paquets de données. Il y a deux ports de lecture de données : un port pour les paquets à retransmettre vers le prochain élément de la chaîne et un port pour les paquets qui ont été acceptés localement. Puisqu'il y a deux ports de lecture des données, il est nécessaire que la mémoire embarquée qui mémorise les données ait également deux ports de lecture et un port d'écriture. Ce dernier fait est important, puisque pour plusieurs plates-formes, une mémoire à deux ports de lecture ne peut être obtenue que par l'utilisation de deux mémoires distinctes. Il existe quatre ports qui permettent d'effacer des paquets : module *Contrôle de flux* lorsqu'un paquet a été envoyé, *Interface/REC* lorsque les données reçues ont été utilisées, *Gestion d'erreur* lorsqu'un paquet a été reçu par erreur et finalement le *Décodeur* lorsque de la corruption des données est détectée.

3.5.6 Tampon de commandes

Une fois que les paquets de commande ont été validés, ils doivent être emmagasinés jusqu'à ce que leur utilisation soit possible. Le module *Tampon de commandes* gère l'enregistrement de ces paquets ainsi que les crédits de contrôle de flux pour les paquets de commande. Ce module est également responsable de déterminer quel est le bon destinataire des paquets ainsi que de réorganiser l'ordre des paquets à transmettre. À cause de la topologie en chaîne de HT, donner priorité à certains types de paquets permet une augmentation de la performance du système. À cause de la complexité importante de ce module, il se sépare en plusieurs sous-modules, tel qu'il est présenté à la figure 3.9.

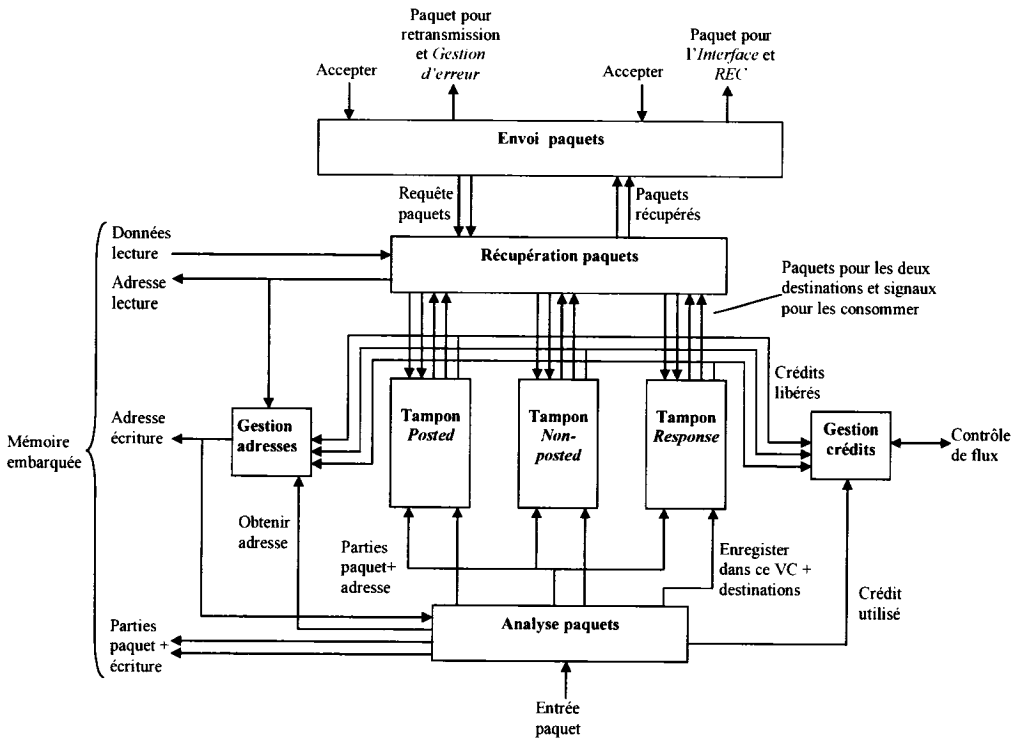

Figure 3.9 - Schéma bloc du *Tampon de commandes*

Un premier sous-module « Analyse paquets » détermine quelle est la destination des paquets reçus et dans quel canal virtuel celui-ci doit être enregistré. Il existe deux

destinations possibles : accepté et non accepté. Un paquet accepté doit se diriger vers le *REC* ou l'*Interface* tandis qu'un paquet non accepté doit se diriger vers le module *Gestion erreur* ou vers le *Contrôle de flux* pour être retransmis au prochain élément de la chaîne. En général, les paquets n'ont qu'une seule destination, à l'exception des paquets de type Broadcast. Une fois que la ou les destinations sont déterminées, le paquet est envoyé au bon canal virtuel avec un code de destination. Deux bits sont utilisés pour ce code de destination et chacun de ces bits représente une destination où le paquet doit se diriger.

Pour respecter les règles du modèle producteur/consommateur tel que défini par la spécification de PCI, un paquet n'a normalement pas le droit de dépasser un paquet de type *posted.* Les applications qui n'ont pas la nécessité de suivre le modèle strict de producteur/consommateur peuvent activer le champ passPW des paquets pour permettre à ces derniers de dépasser les paquets de type *posted.* Ceux-ci ont donc l'avantage d'avoir une plus grande flexibilité et d'êtres transmis à la première occasion, sans devoir attendre que tous les paquets de type *posted* arrivés précédemment aient été transmis.

Puisque les paquets sont enregistrés dans trois canaux virtuels distincts, il faut une méthode pour identifier l'ordre de réception des paquets les uns par rapport aux autres. Afin de garder l'ordre des paquets *posted* par rapport aux paquets *non-posted*, un nombre est donné à tous ces paquets. La valeur de ce nombre est incrémenté lorsqu'un paquet de type *posted* est reçu après avoir reçu un ou plusieurs paquets de type *non-posted* qui n'ont pas le champ passPW activé. Ce nombre permet donc de savoir si un paquet de type *non-posted* est arrivé avant ou après un paquet de type *posted.* Le nombre n'est incrémenté que lors de changements de type de *non-posted* à *posted* pour limiter à $\text{NbBits} = \lceil \log_2(\text{Nombre d'espace tampon par canal virtuel}) \rceil + 1$ le nombre de bits nécessaires pour permettre de maintenir l'ordre entre ces deux types. Un nombre équivalent est utilisé pour maintenir l'ordre entre les paquets *posted* et *response.*

L'utilisation de deux nombres différents pour maintenir l'ordre de réception entre les paquets *non-posted* et *response* par rapport aux paquets *posted* est nécessaire puisqu'il faut considérer qu'un nombre fini de bits doit être utilisé pour ces nombres. Par exemple, si un type de paquet est reçu en grande quantité pendant qu'un paquet d'un autre type demeure dans les tampons, le nombre qui maintient l'ordre du type de paquet reçu en grande quantité peut basculer (*rollover*) plusieurs fois sans invalider celui qui maintient l'ordre de l'autre type de paquet demeuré dans les tampons.

La majorité du contenu des paquets n'est pas nécessaire pour que les canaux virtuels déterminent l'ordre dans lequel les paquets peuvent être envoyés aux deux destinations. Cette partie du contenu peut donc temporairement être enregistrée dans des mémoires embarquées pour minimiser l'utilisation de bascules. Le sous-module Gestion adresse gère les adresses de la mémoire qui sont disponibles pour enregistrer ces parties de paquets. Ce sous-module a une structure interne très similaire au module *Tampon de données* avec des registres qui permettent de mémoriser quelles adresses sont disponibles. Lorsqu'un paquet est reçu, *Analyse paquets* demande une adresse et celle-ci est réservée jusqu'à ce que l'un des canaux virtuels libère un paquet et que le sous-module *Récupération paquets* récupère le paquet à cette même adresse.

Le sous-module *Gestion crédits* mémorise combien d'espace de tampon est disponible dans chaque canal virtuel pour être en mesure d'envoyer le bon nombre de crédits de contrôle de flux au module *Contrôle de flux*.

Le cœur du *Tampon de commandes* est constitué des trois sous-modules *tampon posted*, *tampon non-posted* et *tampon response*. Ceux-ci contiennent les registres qui emmagasinent les paquets. Puisque l'envoi de paquets qui n'ont pas le champ passPW activé est limité par les paquets *posted* ayant été reçus plus tôt, pour améliorer les performances, il est préférable de laisser passer les paquets ayant le champ passPW avant les paquets ayant ce champ désactivé. Pour permettre cela, les sous-modules *tampon* ne

sont pas de simples files d'enregistrement, mais ils permettent de faire une pleine réorganisation des paquets. La figure 3.10 montre la structure interne des sous-modules tampons.

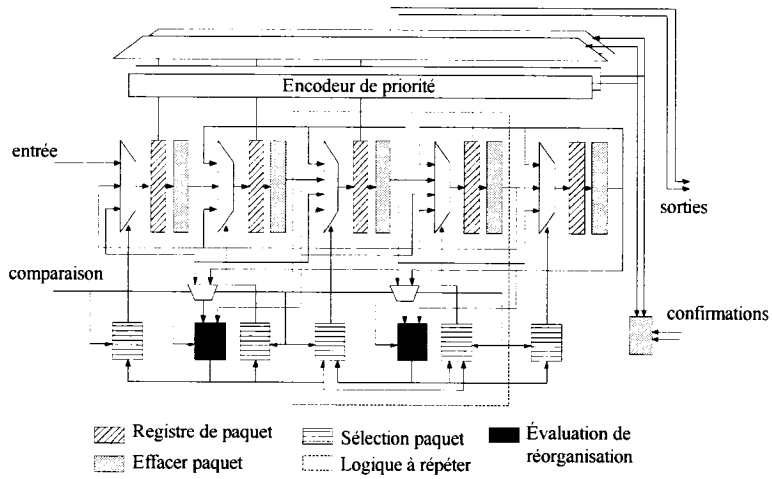

Figure 3.10 - Circuit de réorganisation des paquets

Il est impossible de faire la comparaison de tous les paquets pour choisir le paquet le plus approprié à l'intérieur d'un coup d'horloge, puisqu'une telle comparaison doit examiner chaque paquet de façon séquentielle, ce qui est très long. Cette tactique est celle qui a été utilisée initialement et cela limitait grandement la fréquence d'opération du circuit : la performance était inversement proportionnelle avec la profondeur des tampons. La méthode alternative qui est maintenant utilisée est de réorganiser les paquets à l'intérieur même des tampons.

Lors d'un premier cycle d'horloge, le signal « comparaison » prend la valeur 0 et tous les paquets enregistrés dans les registres pairs sont comparés avec les paquets contenus dans les registres précédents (à gauche dans la figure 3.10). Si le paquet précédent a une priorité supérieure, le contenu des deux registres est échangé. Au prochain coup d'horloge, le signal « comparaison » prend la valeur 1 et les paquets dans les registres pairs sont comparés aux registres suivants (à droite dans la figure 3.10). Cet

algorithme permet donc à un paquet prioritaire de dépasser les autres paquets lorsqu'il passe un certain temps à l'intérieur du tampon. Lorsqu'un nouveau paquet est reçu, tous les paquets se déplacent d'une position pour laisser une place au nouveau paquet et le signal « comparaison » n'est pas modifié. Il est également important de noter qu'il est tenu compte du nombre de fois qu'un paquet a été dépassé par d'autres pour empêcher que ceux avec les bits passPW désactivé ne restent indéfiniment à l'intérieur du tunnel si le réseau est congestionné.

Les encodeurs de priorité permettent de trouver la position des paquets qui se situent à la position la plus profonde dans le tampon pour chacune des deux destinations. Des multiplexeurs viennent alors sélectionner ces paquets et les envoyer vers la sortie. Lorsqu'un paquet pour une destination précise est consommé par le sous-module de récupération de paquet, le bit correspondant à cette destination dans le registre de paquet est mis à 0 (logique « Effacer paquet »). Lorsque les deux bits de destination sont à 0, le paquet a été effacé.

Les paquets qui sortent des tampons ne sont pas complets, puisque plusieurs des champs de ceux-ci se trouvent encore dans des mémoires embarquées. Le sous-module de récupération de paquet fait la lecture d'un paquet en provenance des tampons pour chaque destination, selon ce qui est demandé par le sous-module d'envoi. Il récupère ensuite les champs manquants de ces paquets de la mémoire embarquée pour reconstituer les paquets originaux.

Le sous module *Envoi paquets* a comme rôle de sélectionner quel paquet doit être envoyé à chacune des deux destinations. Une première destination est l'*Interface/REC* tandis que la deuxième est le module *Contrôle de flux* de l'autre côté du tunnel/module *Gestion erreur*. Il est responsable de choisir entre les trois différents types de paquets à envoyer à ces deux destinations. Pour permettre de faire un choix approprié, ce module doit maintenir des registres pour garder jusqu'à 2 paquets de chaque canal virtuel, et cela

pour chaque destination pour un total de 12 registres de paquets. Ces registres occupent une part importante de la complexité du *Tampon de commandes*. La raison pourquoi il est nécessaire d'avoir à emmagasiner plus d'un paquet à la fois est de pouvoir absorber la latence requise pour obtenir un paquet des tampons, puisqu'une partie des paquets doit être récupérée dans une mémoire embarquée.

Le choix du paquet à envoyer se fait selon une liste de priorités préétablie en tenant compte des règles de réorganisation de HT et de l'ordre d'arrivée des paquets les uns par rapport aux autres. Pour empêcher qu'un type de paquet monopolise entièrement le lien de communication, pour chaque type de paquet, un compteur est incrémenté à chaque fois qu'un autre type de paquet est envoyé. Lorsque le compteur arrive à une valeur précise, un changement de priorité a lieu et le paquet peut alors être transmis.

3.5.7 Module *Gestion erreur*

Deux cas peuvent survenir où il faut traiter des paquets reçus par erreur. Le premier survient lorsque l'on reçoit un paquet avec un mode d'adressage sur 64 bits, puisque cette fonctionnalité n'est pas supportée dans l'implémentation courante du tunnel et le deuxième lorsqu'un côté du tunnel n'est pas connecté et qu'un paquet à retransmettre est reçu. Lorsque ces cas surviennent, il n'est pas possible de tout simplement ignorer le paquet car, lorsqu'un paquet de requête de type *non-posted* est envoyé, des ressources sont réservées pour ce paquet jusqu'à ce qu'une réponse soit reçue. Il est donc absolument nécessaire de toujours générer une réponse à une requête *non-posted* en plus d'effacer les données qui lui sont associées et enregistrées dans le module *Tampon de données*. Dans le cas de requêtes *posted* qui sont reçues par erreur, il suffit tout simplement d'effacer les données qui lui sont associées. C'est le rôle du module *Gestion erreur* de traiter tous ces cas.

La conception de ce module est très simple, ses tâches étant très limitées. Il s'agit tout simplement d'un registre d'entrée pour faire la lecture et mémoriser les champs

nécessaires des paquets à traiter, d'une machine à états pour générer les bonnes réponses et permettre d'interagir avec le *Tampon de données* pour effacer les données ainsi que d'un registre de sortie pour envoyer la sortie au module *Contrôle de flux*.

3.5.8 Registres d'espace de configuration

Chaque élément à l'intérieur d'une chaîne HT peut supporter différentes fonctionnalités du protocole, telles que différentes tailles de lien, différentes fréquences d'opération ainsi que nécessiter une différente plage d'adresse mémoire du système. Le module *Registres d'espace de configuration* (*REC*) contient un espace mémoire de configuration qui permet à la fois d'afficher au reste du système les capacités et les besoins en espace mémoire du tunnel tout en permettant à l'hôte de la chaîne d'effectuer la configuration des fonctionnalités. Un aperçu de la structure du *REC* est montré à la figure 3.11.

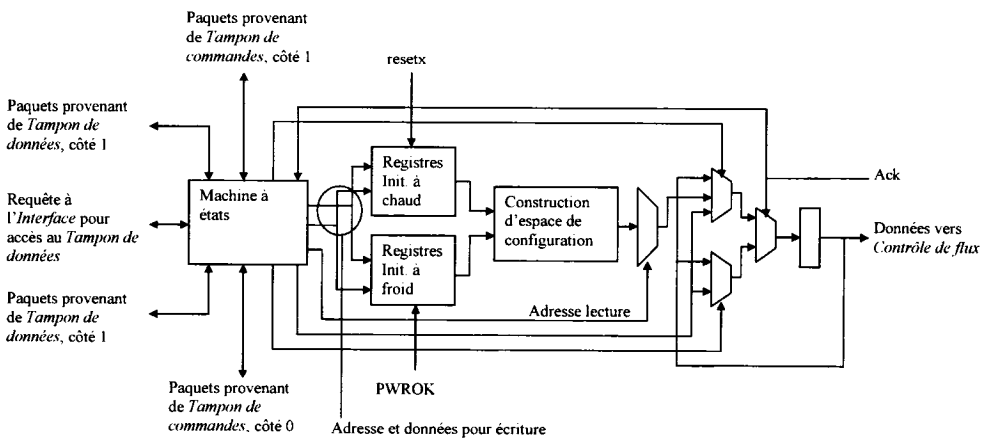

Figure 3.11 - Schéma bloc des *Registres d'espace de configuration* (*REC*)

Les registres sont séparés en deux grandes catégories, ceux à initialisation à froid et ceux à initialisation à chaud. Les registres à initialisation à froid ne sont initialisés que lors du démarrage du circuit à l'aide du signal PWROK, tandis que les autres registres sont plutôt contrôlés par le signal d'initialisation normal. La partie construction d'espace de configuration prend la sortie de tous les registres et les assemble pour qu'ils donnent

un espace de mémoire continu que l'on peut facilement accéder à l'aide d'une adresse envoyée à un multiplexeur.

Le *REC* reçoit des requêtes provenant du *Tampon de commande* de chaque côté du tunnel et traite ces requêtes en effectuant des écritures ou des lectures dans les registres, selon la nature de la requête. Pour effectuer une écriture, il est nécessaire d'aller récupérer les données dans le *Tampon de données*. Il faut en premier lieu demander à l'*Interface* d'avoir accès au *Tampon de données*, puisque son accès est partagé entre ces deux modules. Si une réponse à la requête doit être envoyée, celle-ci est envoyée au module de *Contrôle de flux* du côté du tunnel d'où provenait la requête. La confirmation que le *Contrôle de flux* accepte les données qu'on lui envoie arrive très tardivement dans le cycle d'horloge; c'est pourquoi le *REC* prévoit la prochain sortie à la fois pour le cas où les données sont acceptées où si elles ne le sont pas, afin de minimiser le délai pour calculer la prochaine valeur des données à envoyer.

3.5.9 Interface

À cause de la structure même du tunnel, il existe deux directions par lesquelles il est possible de recevoir et d'envoyer des paquets : vers le haut ou vers le bas de la chaîne. Le module *Interface* permet de cacher cette complexité à l'utilisateur en jouant le rôle d'arbitre entre les deux côtés pour la réception de paquets ainsi qu'en déterminant automatiquement vers quelle direction un paquet doit être envoyé. Tel qu'on peut le voir à la figure 3.12, l'*Interface* est divisée en deux grandes fonctionnalités : RX qui effectue la gestion de la réception des paquets et TX qui effectue la gestion de l'envoi de ceux-ci.

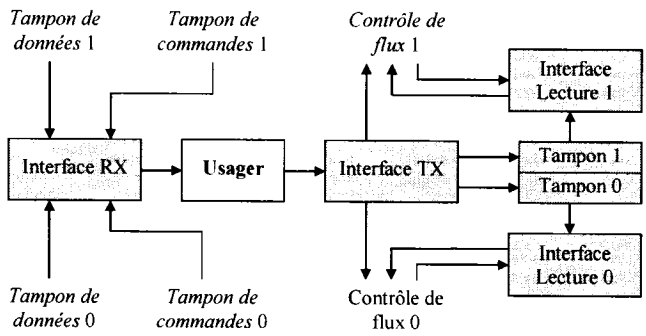

Figure 3.12 - Schéma bloc de l'*Interface*

L'interface RX redirige les paquets provenant des deux modules *Tampon de commandes* vers l'utilisateur en alternance. Lorsque l'usager accepte un paquet de commande qui lui est envoyé, l'interface RX va ensuite chercher les données qui sont associées à ce paquet dans les modules *Tampon de données* et les fait également parvenir à l'utilisateur. Il arrive que le module *REC* ait également besoin d'accéder aux données d'un côté du tunnel et, dans ce cas, l'interface RX se limite à retransmettre des paquets provenant du côté opposé du tunnel à l'usager.

L'interface TX, pour sa part, effectue l'analyse de paquets en provenance de l'usager pour l'envoyer vers la bonne direction. Par défaut les paquets sont dirigés vers le haut de la chaîne, ce qui est déterminé lors de la configuration du *REC*, mais la fonction DirectRoute peut faire en sorte qu'un paquet soit envoyé dans la direction opposée. Normalement, lorsque deux éléments d'une chaîne veulent communiquer entre eux, les paquets sont envoyés vers le haut de la chaîne et l'hôte effectue la réflexion de ces paquets. Avec la fonctionnalité DirectRoute, il est possible d'éviter la réflexion des paquets et de communiquer directement avec un autre élément d'une chaîne HT.

Une fois que la direction vers laquelle un paquet doit se diriger est déterminée, le paquet est transmis au module de *Contrôle de flux* correspondant. Les données qui suivent sont ensuite emmagasinées dans des tampons. Le module de *Contrôle de flux* pourra alors dans le futur venir récupérer ces données. La raison pourquoi il s'agit du

module d'*Interface* qui emmagasine les données est discutée plus en détail dans la section suivante qui porte sur le module *Contrôle de flux*.

3.5.10 Contrôle de flux

Il existe dans le tunnel plusieurs sources pouvant générer des paquets et il y a plusieurs facteurs à considérer pour faire un choix judicieux du prochain paquet à envoyer au prochain élément dans une chaîne HT. Le rôle premier du module de *Contrôle de flux* est d'effectuer la gestion de ce qui doit être transmis. Son rôle ne s'arrête par contre pas là, puisqu'il permet également d'emmagasiner tous les paquets transmis pour permettre de les retransmettre en cas d'erreur de transmission. Il agit également comme tampon pour les paquets en provenance de l'utilisateur. Une version initiale de ce module a été le sujet d'un projet de maîtrise-cours [18]. La figure 3.13 donne un aperçu de sa structure interne.

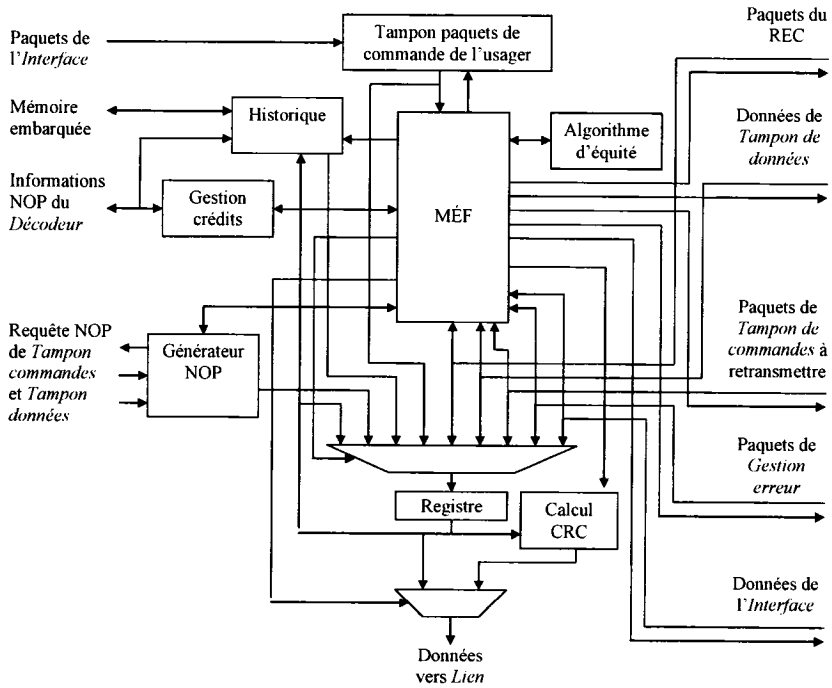

Figure 3.13 - Schéma bloc du module *Contrôle de flux*

Le tampon de paquets de commande provenant de l'usager contient un certain nombre d'espaces réservés pour chaque canal virtuel. Le rôle premier de ce tampon est d'emmagasiner les paquets provenant de l'usager de sorte que si le prochain élément de la chaîne HT ne peut accepter des paquets d'un canal virtuel en particulier, que cela n'empêche pas l'envoi de paquets d'autres canaux virtuels. Un deuxième rôle de ce module est d'absorber la latence de la communication avec l'usager : à cause des délais de propagation de la logique et du temps requis pour que l'*Interface* décide de quel côté du tunnel un paquet sera envoyé, cela peut prendre plusieurs cycles d'horloge avant que l'usager réagisse au fait que l'on ne veut plus recevoir de paquets supplémentaires.

La profondeur des tampons est paramétrable en fonction des diverses options qui affectent la latence de la communication avec l'usager. Les paquets sont enregistrés dans des registres, mais ceux-ci ne permettent pas aux paquets d'être réorganisés à l'intérieur même d'un même canal virtuel. L'ordre dans lequel les paquets sont arrivés en fonction des paquets de type *posted* est enregistré pour s'assurer que les règles de dépassement de paquets de HT ne sont pas enfreintes. La raison pourquoi ces tampons ont été conçus avec des registres et que les données associées aux paquets enregistrés dans ces tampons sont emmagasinés dans l'*Interface* provient du fait que son rôle d'absorption de latence n'avait pas originellement été considéré et que ce tampon devait donc avoir une profondeur très faible.

Le sous-module de gestion de crédit reçoit l'information de contrôle de flux contenue dans les paquets NOP, ce qui permet de savoir combien d'espace tampon est disponible dans l'élément de la chaîne HT pour lequel le module *Contrôle de flux* envoie des paquets. Il met à jour la quantité d'espace de tampon qui est disponible à chaque fois qu'un paquet est envoyé. Le générateur NOP considère l'espace tampon qui est disponible dans le tunnel et génère des NOP afin de transférer l'état des tampons du tunnel au prochain élément de la chaîne HT.

Lorsque le mode *retry* est activé, le sous-module de calcul de CRC génère un code CRC pour chaque paquet de commande transmis. Le sous-module d'historique garde en mémoire tous les paquets envoyés, excluant les paquets NOP, jusqu'à ce que leur bonne réception soit confirmée. Si une erreur de transmission survient, il est alors possible de réinitialiser la communication et de retransmettre les paquets n'ayant pas été reçus correctement par le prochain. Les paquets NOP, en plus de contenir de l'information de contrôle de flux, contiennent également un champ qui représente le numéro d'identification du dernier paquet reçu correctement. Lorsqu'un paquet NOP est reçu avec un tel numéro, il est donc possible de l'éliminer de l'historique ainsi que tous ceux ayant été envoyés avant celui-ci.

L'historique des paquets transmis est enregistré dans une mémoire embarquée. Initialement, la taille et la position de chaque entrée de l'historique étaient enregistrées dans des registres. Considérant que pour chaque paquet transmis, un code CRC de 32 bits est envoyé, il par contre possible d'utiliser le temps où le code est transmis pour enregistrer dans la mémoire embarquée le numéro d'identification du paquet, qui permet ensuite de l'effacer lorsqu'on a la confirmation qu'il a été reçu correctement par le prochain élément de la chaîne, ainsi que la taille du paquet. Cette méthode permet d'éliminer une quantité importante de bascules dans le tunnel.

Au coeur du *Contrôle de flux*, une machine à états effectue la décision de ce qui doit être envoyé au lien à chaque cycle d'horloge. Cette machine à états qui compte plus de 40 états, lorsque le mode *retry* est implémenté, effectue l'analyse de tous les paquets disponible de la part du *REC*, du module *Gestion erreur*, des tampons de paquets de l'usager, des paquets provenant du *Tampon de commandes* de l'autre côté du tunnel et finalement du générateur de NOP. En plus de tout cela, la machine à états doit également gérer les données qui sont associées aux paquets de commandes, la retransmission de paquets lors d'une séquence de *retry*, envoyer les codes CRC à la suite des paquets

lorsque le mode *retry* est activé et également ne pas modifier la sortie lorsque le lien n'effectue pas la lecture les données.

À cause de la topologie en chaîne de HT, il est nécessaire de porter une attention particulière sur la méthode d'insertion des paquets d'origine locale dans le flux de paquets retransmis. Si une grande quantité de paquets étaient insérés par chaque élément de la chaîne, un élément à la fin de la chaîne n'aurait plus qu'une très petite fraction de la bande passante. Pour éviter cette situation, un algorithme d'équité est établi par la spécification de HT 2.00b, section 4.9.5, p. 73. Cet algorithme, qui est pleinement implémenté dans le tunnel, assure une bande passante équitable à tous les éléments d'une chaîne.

3.6 Comparaison aux architectures commerciales actuelles

Il existe plusieurs circuits de contrôle de communication HyperTransport disponibles commercialement [1][34][12][13][14] et il est donc intéressant de comparer le tunnel présenté dans ce mémoire avec ces implémentations. Un des atouts du tunnel est qu'il est la première implémentation connue du mode *retry*. Il n'y a également aucune indication précise que les designs présentement sur le marché effectuent une réorganisation des paquets à l'intérieur de leurs canaux virtuels. La majorité des circuits HT utilisent un FIFO pour absorber les variations de domaines d'horloge, comme il est expliqué à la section 3.5.1; il s'agit d'une méthode différente de celle utilisée par le tunnel. Les implémentations commerciales atteignent par contre une plus grande bande passante en utilisant des chemins de données de 64 ou même 128 bits au lieu de 32 bits. Les implémentations commerciales supportent également un lien de 16 bits en général, ce qui n'est pas le cas du tunnel présenté ici.

3.7 Conclusion

Afin de partager les résultats de conception d'un tunnel HT, le tunnel a été publié sur le site web opencores.org sous la licence libre Mozilla Public Licence (MPL). Le tunnel comporte de nombreux paramètres qui permettent de l'adapter en fonction des besoins de l'application. Ceci est permis par le précompilateur C++ qui peut être utilisé d'une multitude de façons pour permettre de modifier le circuit en fonction des paramètres désirés.

Tel qu'il est mentionné dans le présent chapitre, le tunnel est divisé en 8 grands modules : *Lien* qui gère l'initialisation de la communication, *Tampon de commandes* qui accumule les en-têtes des paquets reçus, *Tampon de données* qui accumule les données reçues, *Gestion erreur* qui traite les paquets reçus en erreur, *Interface* qui permet au tunnel de communiquer avec un utilisateur, *REC* qui contient une multitude de registres pour afficher et configurer les paramètres du tunnel et finalement *Contrôle de flux* qui gère les paquets à envoyer. À cause de leur complexité, plusieurs de ces modules ont été divisés en plusieurs sous–modules, d'ailleurs présentés dans ce chapitre.

Chapitre 4. Résultats de synthèse et vérification du tunnel

Un des buts premiers de cette recherche est d'approfondir nos connaissances sur la performance et la complexité des protocoles de communication interpuces. Il est très difficile d'évaluer la complexité qu'aura un circuit qui met en œuvre un protocole à l'aide uniquement de la spécification des fonctionnalités de ce protocole. Les résultats de synthèse du tunnel HT permettent d'obtenir des données quantitatives sur les ressources que nécessitent les diverses parties du circuit, ainsi que le coût associé aux divers paramètres du circuit. À partir de ces informations, il est possible de déterminer quelles améliorations sont possibles pour diminuer cette complexité et améliorer la performance. Certaines améliorations ont déjà été effectuées et celles-ci sont traitées. Un aspect également très important lors de la conception d'un circuit est la vérification de celui-ci pour s'assurer de son bon fonctionnement. La vérification occupe une part très importante du temps de développement et la méthode utilisée pour faire la validation du tunnel est présentée dans le présent mémoire.

4.1 Données de synthèse

La synthèse du tunnel HT a été effectuée pour la technologie de cellules normalisées CMOS 0,18 µm de TSMC à l'aide de Synopsys Design Compiler, pour la technologie FPGA Virtex II Pro de Xilinx et Stratix d'Altera à l'aide de Synplify. Dans tous ces cas, le code source SystemC a été traduit vers le Verilog à l'aide de l'outil Synopsys SystemC Compiler avant d'être traité par les outils de synthèse. La majorité des données sont présentées pour la technologie de cellules normalisées afin de simplifier la présentation des données et puisque l'outil de synthèse permet d'obtenir plus facilement des données sur la distribution de la complexité à l'intérieur d'une hiérarchie de composants. La quantité de ressources nécessaires pour la mise en œuvre du tunnel

pour cette technologie est présentée en milliers de portes logiques. Elle est calculée en prenant la superficie totale des cellules composant le circuit (excluant le routage) et en divisant celle-ci par la taille d'une porte logique NAND à 2 entrées de taille minimale.

En premier lieu, le tableau 6 présente l'utilisation des ressources et la performance du tunnel pour les diverses technologies considérées. Ces résultats ont été obtenus pour un tunnel qui comporte des tampons de commandes d'une profondeur de 7 paquets et des tampons de données d'une profondeur de 8 paquets. Dans le cas de la technologie de cellules normalisées, la fréquence d'opération présentée correspond au cas où le procédé de fabrication donne la pire performance possible. Le tableau présente également la bande passante du design. Celle-ci n'est pas directement proportionnelle avec la fréquence d'opération maximale du circuit puisque le tunnel doit opérer uniquement aux fréquences permises par la spécification HT. Ces données n'incluent pas les mémoires embarquées du tunnel, qui totalisent 43,2 Kibit (1024 bits) en total, distribué entre les deux modules *Tampon de données* (24 Kibit), les deux modules *Tampon de commandes* (3,2 Kibit), pour garder l'historique des paquets envoyés (8 Kibit) et pour garder en tampon les paquets de l'utilisateur à envoyer (8 Kibit).

Tableau 6 - Données de synthèse du tunnel HT pour diverses technologies

	Ressources	**Fréquence (MHz)**	**Bande passante (Mb/s)**
0.18µm cellules standards (pire cas)	131,9 kportes	251,3*	250 * 32 = 8000
Xilinx FPGA Virtex II Pro (-7)	18,1 kLUTs	103,7	100 * 32 = 3200
Altera FPGA Stratix (-5)	21,9 kLEs	88,4	75 * 32 = 2400

* Avant placement et routage

4.1.1 Distribution de la complexité

Il est assez difficile de bien comprendre d'où vient la complexité des circuits de contrôle pour les protocoles de communication interpuces. La présentation suivante sur la distribution de la complexité logique entre les divers éléments du tunnel permet de bien cibler les raisons qui occasionnent une telle complexité. La distribution de complexité à l'intérieur de chaque module, qui comporte des sous-modules, est également présentée ci-dessous.

La figure 4.1 nous montre la distribution des portes logiques à l'intérieur du tunnel. En observant ce graphique, il est important de noter que tous les modules, sauf *REC* et *Interface*, sont présents à deux reprises dans le tunnel. Il est possible de remarquer que ce sont les modules *Tampon de commandes* et *Contrôle de flux* qui occupent la majorité des ressources du tunnel. Il est normal que le T*ampon de commandes* nécessite beaucoup de ressources puisqu'il doit contenir une grande quantité de bascules pour enregistrer les paquets. Du côté du *Contrôle de flux*, une inspection de ses sous-modules permettra de mieux identifier la source de sa complexité.

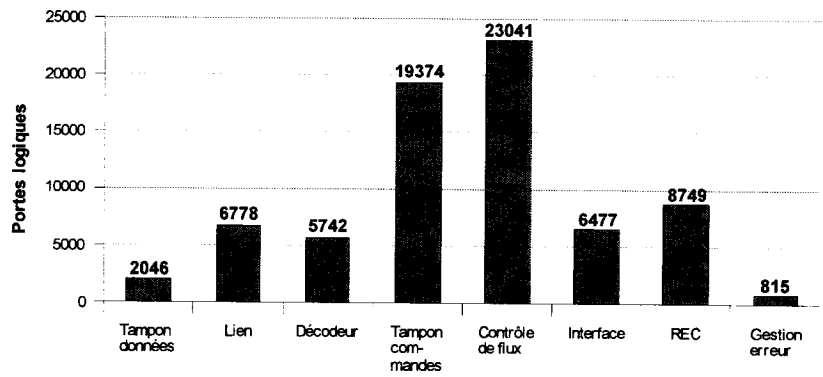

Figure 4.1 - Répartition de la complexité du tunnel entre les modules

Le module *Lien* occupe un espace non négligeable, surtout lorsque l'on considère son rôle limité d'effectuer l'initialisation de la communication. La répartition de la

complexité à la figure 4.2 nous permet d'identifier que presque trois quarts de la complexité provient de la génération de CRC périodique du lien.

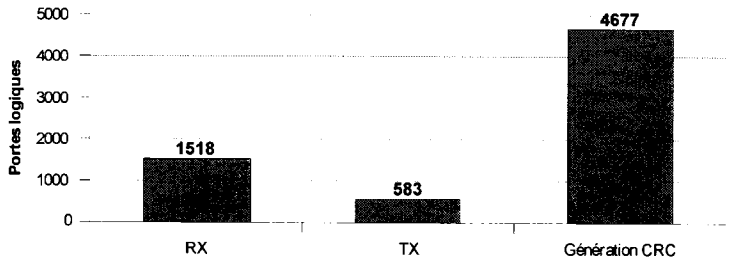

Figure 4.2 - Répartition de la complexité à l'intérieur du module *Lien*

Du côté du *Décodeur*, c'est encore une fois la vérification de CRC qui occupe la majorité de l'espace. Les registres pour emmagasiner les paquets occupent également un espace non négligeable.

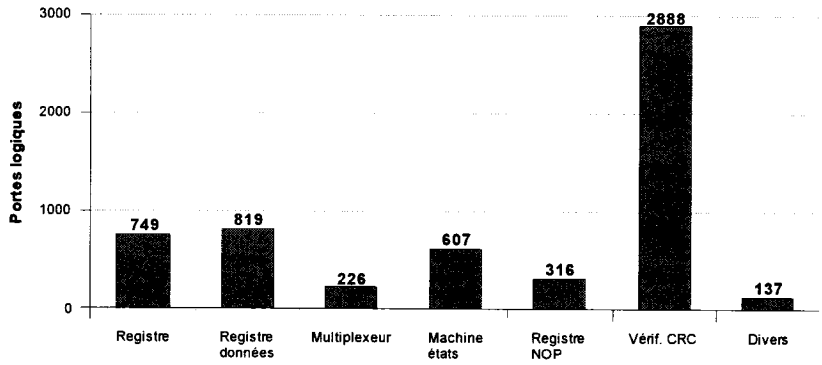

Figure 4.3 - Répartition de la complexité à l'intérieur du module *Décodeur*

Il est très intéressant de se pencher sur le module *Tampon de commandes*, puisque celui-ci occupe un espace très important à l'intérieur du tunnel. La première chose que l'on peut remarquer est que l'utilisation de ressources est dominée par le sous-module d'envoi avec 11,3 kportes. Cette importante utilisation de ressources est expliquée par le fait que ce module tamponne deux paquets de chaque canal virtuel, et cela pour les deux destinations possibles pour un total de 8 registres de paquets de 64 bits et de 4 paquets de 32 bits.

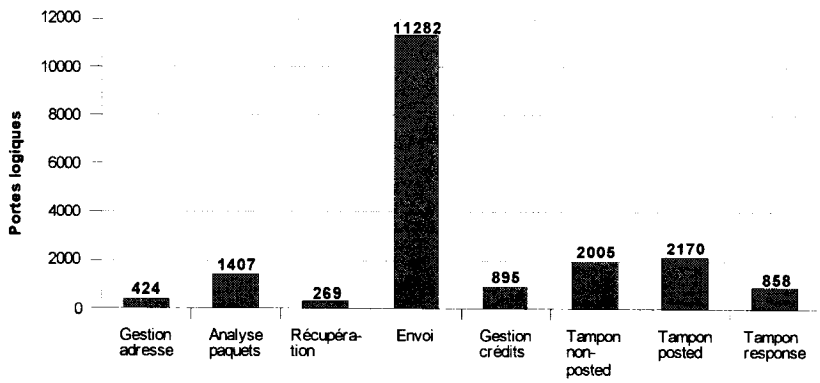

Figure 4.4 - Répartition de la complexité à l'intérieur du module *Tampon de commandes*

La raison pourquoi le sous-module d'envoi tamponne 2 paquets de chaque sorte est pour absorber une partie de la latence nécessaire pour récupérer des paquets de différents canaux virtuels, puisque ceux-ci sont en partie emmagasinés dans des mémoires embarquées. Une solution qui permettrait de réduire la complexité du sous-module d'envoi serait que le sous-module de récupération de paquet ne fasse son travail que lorsque le module d'envoi décide qu'un paquet est prêt à être envoyé. Le sous-module d'envoi de paquets n'aurait donc qu'à enregistrer les fractions de paquets nécessaires pour faire l'évaluation du prochain paquet à envoyer. Il serait également possible de modifier le design afin d'avoir un tampon d'une profondeur de 1 paquet.

Les tampons *non-posted*, *posted* et *response* ont une taille très raisonnable, du fait qu'ils ne contiennent que les parties des paquets nécessaires à évaluer la réorganisation de l'ordre des paquets. La taille de ces sous-modules varie avec la profondeur des tampons.

Le module de *Contrôle de flux* souffre d'un problème très similaire à celui du *Tampon de commandes*, un des sous-modules occupe une très grande part de ses ressources comme le montre la figure 4.5. Le sous-module Tampon utilisateur accumule les paquets de l'utilisateur avant que ceux-ci ne soient envoyés au *Lien*. Ce tampon doit avoir une profondeur suffisante pour absorber la latence de la communication avec

l'utilisateur, qui peut être de plusieurs cycles d'horloge. Une façon de diminuer significativement la complexité de ce module serait d'enregistrer les paquets dans une mémoire embarquée au lieu d'utiliser des bascules.

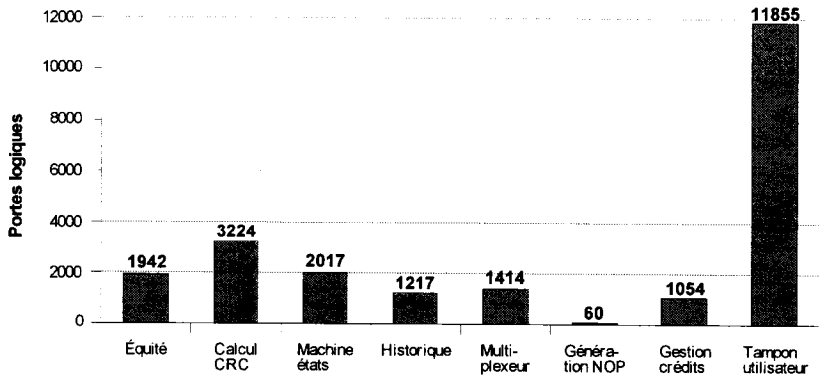

Figure 4.5 - Répartition de la complexité à l'intérieur du module *Contrôle de flux*

Le reste de la complexité du *Contrôle de flux* est répartie entre ses divers autres sous-modules. On peut noter que le calcul de CRC occupe encore une fois une complexité relativement importante.

4.1.2 Effets des paramètres de synthèse

Pour permettre un maximum de flexibilité, le tunnel comporte une série de paramètres qui ont un impact sur son utilisation de ressources. L'option qui a le plus d'impact sur le tunnel est de supporter ou non le mode *retry*. Deux autres options qui ont un léger impact sur la complexité est d'effectuer une correction interne d'alignement si le désérialiseur externe ne peut le faire et de supporter ou non le mode DirectRoute. Le tableau 7 résume l'effet de ces options sur la complexité du tunnel. De plus, d'autres options telles que la latence de la communication avec l'utilisateur (qui affecte la profondeur du tampon du Contrôle de flux), la profondeur du *Tampon de données*, la profondeur du *Tampon de commandes* et la réorganisation des paquets à l'intérieur des canaux virtuels.

Tableau 7 - Effet des paramètres du tunnel sur les résultats

	Portes logiques (x1000)	**Mémoires embarquées utilisées (Kibits)**	**Fréquence d'opération (MHz)**
Options par défaut	131,9	43,2	251,3
Sans le support pour le mode *retry**	111	35,8	266,0
Avec alignement interne*	133,4	43,2	251,3
Sans le support pour le mode DirectRoute*	128,4	43,2	251,3
Sans réorganisation des paquets*	122,8	43,9	251,3

* Avec toutes les autres options par défaut

Il est normal que la taille du tunnel varie de façon importante avec l'inclusion du mode *retry* puisque, lorsqu'il est désactivé, le *Décodeur* et le *Contrôle de flux* n'ont pas besoin d'effectuer de calcul de CRC et ont des machines à états plus simples. Un historique des paquets envoyés n'est également pas nécessaire. Le réalignement interne des données n'ajoute que peu de complexité, puisque cela ne représente qu'un décaleur de bits avec un certain nombre de bascules. L'élimination du mode DirectRoute permet de sauvegarder des registres dans le module *REC* et de la logique de décodage d'adresse dans l'*Interface*. Finalement, ne pas effectuer la réorganisation des paquets épargne une bonne quantité de logique, puisqu'il est alors possible d'envoyer une plus grande partie des paquets de commande vers les mémoires embarquées.

Malgré le fait qu'une bonne partie des paquets sont enregistrés à l'intérieur de mémoires embarquées, le nombre de tampons utilisés dans le design peut toutefois avoir un impact assez important sur la taille du tunnel, tel qu'on peut le voir à la figure 4.6. On peut noter qu'entre une taille de 3 et 5 et entre 7 et 9, la différence de complexité est légèrement plus élevée qu'aux autres intervalles. Cela est causé par le nombre de bits

requis pour représenter l'adresse des tampons qui passe respectivement de 2 à 3 et de 3 à 4 bits.

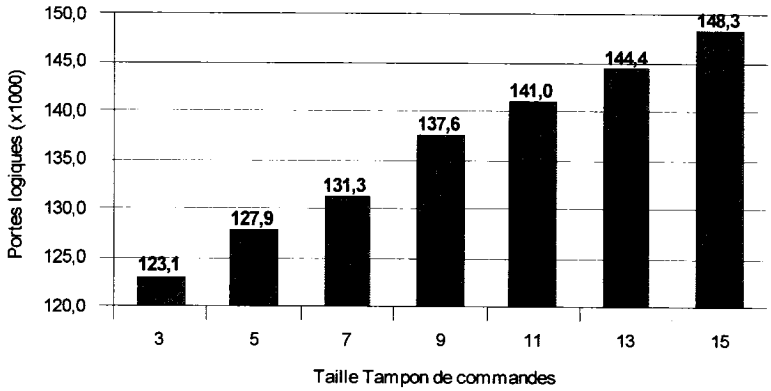

Figure 4.6 - Complexité du tunnel en fonction de la profondeur des tampons de commandes

Il est possible de désactiver la réorganisation de paquets à l'intérieur des canaux virtuels, ce qui affecte de façon significative la complexité du tunnel (voir la figure 4.7). Cette option permet d'économiser des ressources, mais très peu d'efforts ont été placés à optimiser ce cas. Il serait donc probablement possible d'éliminer encore une plus grande partie de la complexité du *Tampon de commandes* lorsque la réorganisation est désactivée à l'aide d'une meilleure utilisation des mémoires embarquées.

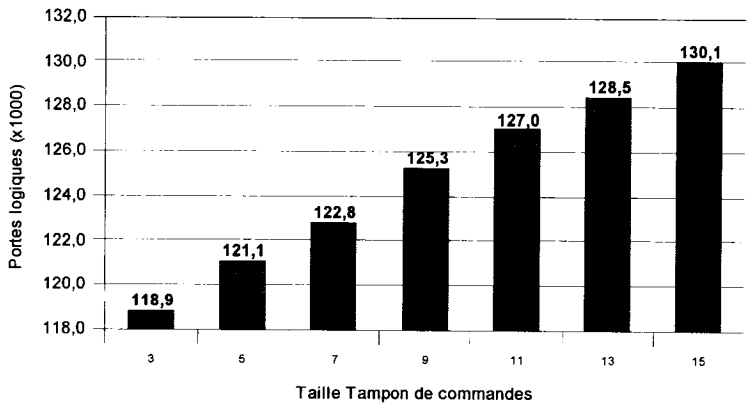

Figure 4.7 - Complexité du tunnel en fonction de la profondeur des tampons de commandes, sans réorganisation des paquets

Pour ce qui est des tampons de données, la figure 4.8 démontre vraiment le nombre de bits d'adresse des tampons de données qui font une différence, puisque la différence de complexité entre les tailles 2 et 4, 4 et 6 et 8 et 10 est beaucoup plus importante que pour les autres différences de taille. Cela est expliqué par le fait que tous les registres de paquets de commandes dans le tunnel contiennent en général aussi l'adresse des données associées à ce paquet, donc si le nombre de bits pour représenter l'adresse d'un paquet de données augmente, la taille des registres de paquet de commandes augmente également à plusieurs endroits. Il est important de noter que la figure 4.8 ne tient pas compte de la variation de la taille des mémoires embarquées.

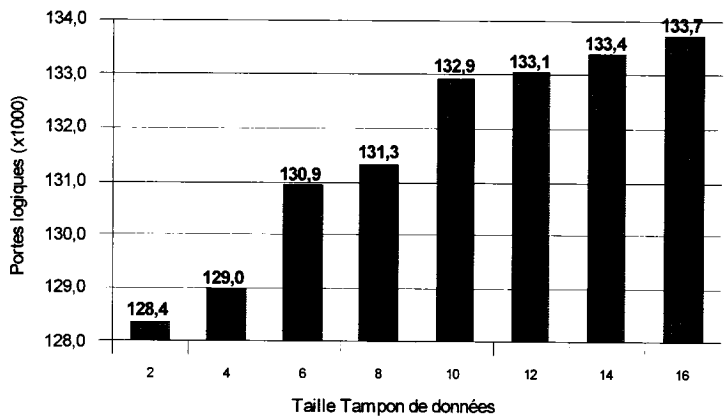

Figure 4.8 - Complexité du tunnel en fonction de la profondeur des tampons de données

Le dernier paramètre que nous allons analyser est la variation de la complexité du tunnel en fonction du nombre de tampons pour emmagasiner les paquets envoyés par l'utilisateur. La figure 4.9 montre l'effet de ce paramètre sur le tunnel. On peut remarquer que celui-ci a un effet très important et c'est pourquoi il serait très intéressant d'emmagasiner ces paquets dans des mémoires embarquées au lieu de les emmagasiner dans des bascules. Il faut noter que la variation de la complexité est très peu linéaire, puisque la taille des tampons change en fonction de la latence de la communication avec l'utilisateur, ce qui est déterminé par l'utilisation ou non de certains registres dans l'*Interface* qui font également varier la complexité.

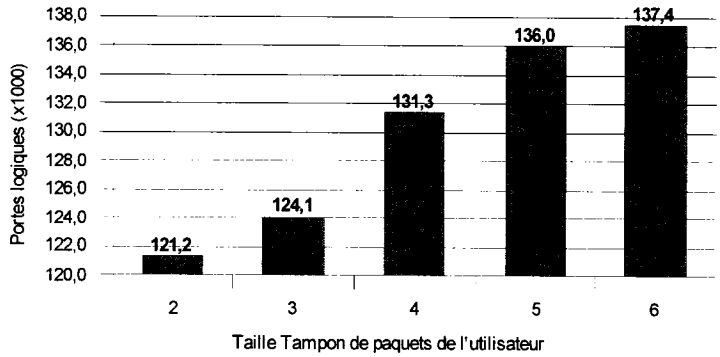

Figure 4.9 - Complexité du tunnel en fonction de la profondeur des tampons de paquets envoyés par l'utilisateur

4.1.3 Chemin critique

Comme tout circuit synchrone, le tunnel HT a une limite de fréquence d'opération en fonction de son chemin critique. Le chemin critique est le calcul qui demande le plus de temps à l'intérieur d'un circuit synchrone et celui-ci est important, puisqu'il détermine la période minimale de l'horloge de synchronisation du circuit. Pour améliorer la performance d'un circuit, il faut donc trouver des façons de réduire ce chemin critique. Dans le cas du tunnel, le chemin critique survient lorsque le module de *Contrôle de flux* effectue la sélection du prochain paquet à envoyer et ce chemin est illustré à la figure 4.10.

En premier lieu, les paquets provenant de diverses sources deviennent disponibles. Le module de *Contrôle de flux* décide ensuite si chacun des paquets peut être envoyé si suffisamment de crédit de contrôle de flux sont disponibles, s'il y a suffisamment d'espace dans l'historique de paquets, si un transfert d'une suite de paquets qui ne doit pas être interrompu est en cours et si l'envoi de ce type de paquet est bloqué par l'algorithme d'équité. Une fois que l'on a déterminé quels paquets peuvent être envoyés, il faut ensuite décider lequel de ceux-ci sera envoyé en considérant l'algorithme d'équité, la priorité du paquet et les besoins de contrôle de flux. Cette dernière étape est effectuée par une structure similaire à un encodeur de priorité, ce qui demande un temps

relativement élevé. Un signal est ensuite envoyé à la source du paquet sélectionné pour l'avertir que le paquet a été sélectionné et qu'il peut fournir un prochain paquet ou une prochaine donnée.

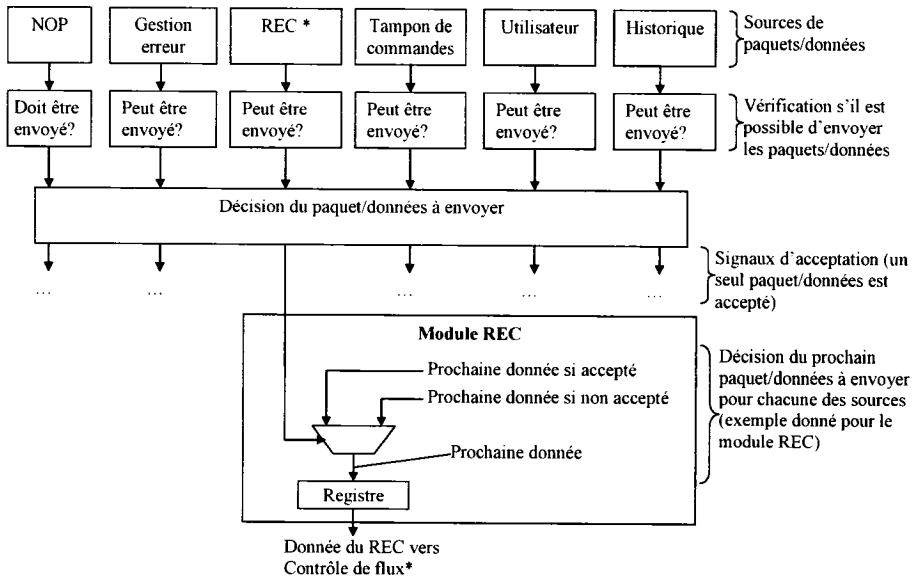

Figure 4.10 - Chemin critique du tunnel

Pour optimiser la performance du circuit, les paquets disponibles sont acheminés le plus rapidement possible à l'entrée du module de *Contrôle de flux*. L'algorithme de décision du paquet à envoyer a également été fortement travaillé pour optimiser sa performance et minimiser les calculs séquentiels. Finalement, la majorité des modules qui envoient des paquets au *Contrôle de flux* prennent d'avance la décision de ce que sera la sortie si elle est acceptée et si elle n'est pas acceptée. Lorsque le signal d'acceptation est disponible, il suffit de faire le choix de la bonne valeur pour la nouvelle sortie. La figure 4.10 montre cette sélection pour le module REC, mais toutes les sources de paquets utilisent cette méthode. La source de paquets ou données qui causent le chemin critique varie d'une fois à l'autre en fonction du routage.

4.2 Amélioration de la complexité et de la performance

Une très grande partie du travail présenté jusqu'à maintenant provient de plusieurs itérations d'un processus de conception pour lesquelles un apprentissage a été effectué. Cette section présente quelques-unes des grandes améliorations qui ont déjà été effectuées ainsi que certaines améliorations à venir.

4.2.1 Lien

Originellement, le *Lien* était conçu pour opérer à la même fréquence que la fréquence d'opération de la couche physique. Ceci était une erreur, puisqu'il est impossible que de la logique, même si elle est relativement simple, puisse fonctionner à une fréquence aussi élevée. Le *Lien* a donc été modifié pour fonctionner à la même fréquence d'opération que le reste du tunnel en traitant les données entrantes et sortantes de façon parallèle.

4.2.2 Tampon de données

Lorsque de petits paquets de données sont reçus, tel qu'un paquet de 4 octets, celui-ci occupe par défaut un espace tampon de 64 octets. Lors de la conception de spécification exécutable de ce module, la décision fut prise de combiner plusieurs petits paquets à l'intérieur d'un même espace tampon pour mieux utiliser l'espace mémoire disponible. De façon logicielle, ceci était très facile à effectuer. Une fois implémentée en matériel, cette fonctionnalité faisait plus que doubler la complexité du *Tampon de données* sans apporter une amélioration de performance pour des raisons discutées au paragraphe suivant et qui n'était pas évidentes a priori. Cette fonctionnalité fut donc éliminée.

Revenons donc sur cette possibilité de compacter plusieurs petits paquets dans un seul espace tampon plus grand. Rappelons que dans le protocole HT, avant d'envoyer un paquet, il faut s'assurer que le prochain élément de la chaîne HT a suffisamment d'espace

tampon. Lorsque l'on envoie deux paquets de 4 octets, selon la spécification HT, il faut considérer que deux espaces tampons seront occupés. Avec la fonctionnalité de regrouper plusieurs petits paquets à l'intérieur d'un même tampon, il était possible d'enregistrer les deux petits paquets à l'intérieur du même tampon. Il était donc possible d'afficher immédiatement que l'on avait un tampon disponible.

Cela peut sembler avantageux au premier regard, mais il faut comprendre que dans HT, le rôle des tampons est d'absorber la latence du lien, de sorte que si les canaux virtuels ne sont pas bloqués, on puisse envoyer des paquets en continu. Dans le cas de la fonctionnalité présentée précédemment avec l'exemple de deux paquets de 4 octets qui n'occupent qu'un seul espace tampon, le fait qu'un espace tampon soit disponible n'est connu que par le récepteur et cette information doit alors être transmise par un paquet de contrôle de flux à l'émetteur, ce qui prend du temps et en fin de compte ne diminue aucunement la latence de la communication et n'offre donc pas d'avantages.

Une autre amélioration qui a été effectuée est qu'originellement, ce module comportait trois ports de lectures. Plus il y a de ports de lecture de données, plus il faut que la mémoire embarquée du *Tampon de données* ait de ports de lecture. Pour réduire le nombre de ports de lecture de la mémoire, le port de lecture du *REC* et celui de l'*Interface* ont été combinés et ces deux modules partagent maintenant un seul port.

4.2.3 Tampon de commandes

Ce module a subi plusieurs grands changements au cours de la conception du tunnel. En premier lieu, les paquets n'étaient par réorganisés à l'intérieur des tampons : les paquets étaient gardés dans l'ordre dans lesquels ils étaient arrivés, mais seul le plus prioritaire était sélectionné. Cette sélection impliquait un algorithme séquentiel et plus la taille du *Tampon de commandes* était élevée, plus la fréquence d'opération du circuit était faible. L'algorithme de réorganisation des paquets a donc été modifié pour éviter un long chemin critique combinatoire.

Deuxièmement, la taille de ce module était très grande. Pour minimiser le nombre de bascules à utiliser dans ce module, il a été décidé d'enregistrer les parties non utilisées pour la réorganisation des paquets à l'intérieur de mémoires embarquées, permettant ainsi d'économiser des ressources logiques. Les paquets sont donc réassemblés juste avant la sélection finale du paquet à envoyer. Une modification future à apporter à ce module pour réduire encore une fois sa complexité serait de n'effectuer le réassemblage des paquets que lorsque ceux-ci sont prêts à être transmis à leur destination finale à l'extérieur du *Tampon de commandes*.

4.2.4 Contrôle de flux

Une des parties qui a le plus évolué lors de la conception de ce module est le tampon d'accumulation de paquets en provenance de l'utilisateur. Originellement, celui-ci ne devait servir qu'à accumuler un seul paquet de chaque type pour permettre d'effectuer un meilleur choix de paquets à envoyer entre les trois canaux virtuels. Ce n'est que plus tard, lors d'optimisations de la performance du circuit qu'il a été réalisé que celui-ci devrait également absorber la latence de la communication avec l'utilisateur, et devrait donc avoir une profondeur relativement importante. Ce module occupe donc maintenant une partie très considérable du tunnel. Une modification bénéfique à apporter serait d'accumuler les paquets à l'intérieur de mémoires embarquées pour permettre de réduire l'utilisation de bascules.

4.2.5 Augmentation de la taille du chemin de données

Il est improbable qu'il soit possible d'augmenter de façon significative la fréquence d'opération du circuit à cause des limites propres des technologies utilisées. Afin d'augmenter la bande passante du tunnel, il faut donc considérer augmenter la quantité de données traitées à chaque cycle d'horloge en augmentant la taille du chemin de données. Le tunnel traite présentement en parallèle 32 bits par cycle d'horloge et il serait possible d'augmenter cela à 64 bits ou même plus.

Un tel changement ne serait pas extrêmement complexe, mais nécessiterait une multitude de changements partout dans le tunnel. Le changement le plus important se situerait au niveau de la réception des paquets, puisque la tâche de décodage, de validation et d'enregistrement des paquets devient plus complexe plus il y a de données à traiter à la fois. La complexité provient surtout du fait que la taille minimale des paquets HT est de 32 bits et qu'il est donc possible que 64 bits et plus de données contiennent plusieurs de ces paquets de 32 bits. Cela pourrait entre autres complexifier le module de *Tampon de commandes* de façon significative.

4.3 Vérification du tunnel

Vérifier le bon fonctionnement d'un circuit est une tâche très ardue, puisque pour effectuer cette vérification, il faut générer des vecteurs d'entrée pour le circuit et, par la suite, vérifier les sorties de ce même circuit pour s'assurer qu'il répond correctement aux stimuli qu'on lui a fournis. Normalement, une telle tâche occupe souvent plus des deux tiers du temps de développement d'un système complexe de qualité industrielle. Dans le cas du tunnel, cela est peu réaliste dans un contexte académique considérant le temps restreint d'une maîtrise, et le temps consacré à la vérification a été limité à environ un quart du temps de développement.

Une approche ascendante a été utilisée pour effectuer la vérification du tunnel. En premier lieu, des bancs d'essai ont été construits pour les sous-modules importants. Une fois que leur bon fonctionnement a été vérifié, des bancs d'essai ont ensuite été construits pour les modules et finalement pour le tunnel en entier.

Les bancs d'essai sont conçus en C++ à l'aide de la bibliothèque SystemC. Le fait que le tunnel est également écrit en SystemC a grandement facilité la vérification, puisqu'un simple compilateur C++ était nécessaire, ce qui est facilement disponible gratuitement sur une variété de plateformes. Une façon commune de produire des stimili dans un banc d'essai consiste à générer des entrées de façon aléatoire et de vérifier que

les sorties sont correctes. Quelques bancs d'essai qui ont été conçus plus tôt dans le cycle de développement n'effectuent pas de vérification des sorties : une vérification visuelle était donc nécessaire, mais ces cas sont très limités. Le design comporte un total de 13 bancs d'essai et ils utilisent une bibliothèque partagée pour faciliter la vérification.

4.4 Conclusion

La synthèse du tunnel HT a permis de déterminer la complexité logique et la performance de celui-ci pour une variété de plates formes telles que la technologie de cellules standards et les FPGA de Xilinx et Altera. Le tunnel, avec les options par défaut, a une complexité de 131,9 milliers de portes logiques. Il nécessite 43,2 Kibits de mémoire embarquée, ce qui n'est pas inclus dans le calcul de la complexité. Il a une bande passante qui va de 2400 Mb/s lorsque la logique est synthétisée pour un FPGA Stratix (-5) d'Altera alors qu'il peut atteindre jusqu'à 8000 Mb/s lorsqu'il est synthétisé pour la technologie de cellules normalisée CMOS 180 nm de TSMC (Taiwan Semiconductor Manufacting Corporation). Le tunnel a été vérifié à l'aide d'une approche ascendante en utilisant le langage C++ avec la bibliothèque SystemC.

Une grande majorité de la complexité du tunnel provient de tampons qui permettent d'emmagasiner des paquets à l'intérieur de registres. Le *Tampon de commandes* et le *Contrôle de flux*, parce qu'ils contiennent tous deux un grand nombre d'espaces tampons, occupent respectivement 29,4 % et 34,9 % du tunnel. Si l'on considère uniquement les tampons qui permettent d'accumuler les paquets envoyés de l'utilisateur, ceux-ci occupent 17,9 % du tunnel Ces derniers tampons sont nécessaires afin d'absorber la latence de la communication avec l'utilisateur. Dans le cas du *Tampon de commandes*, les tampons sont nécessaires pour absorber la latence du lien de communication et certains tampons sont nécessaires pour absorber la latence des mémoires embarquées. Dans les modules *Lien*, *Décodeur* et *Contrôle de flux*, le calcul de CRC prend également une quantité de logique non négligeable.

Plusieurs paramètres peuvent affecter le tunnel HT et ceux-ci font varier la performance et la complexité du tunnel. L'élimination du support pour le mode *retry* permet d'économiser 20,9 milliers de portes logiques et permet d'augmenter la fréquence d'opérations de 5,8 %. Les autres options telles que l'alignement interne des données et le support pour le mode DirectRoute n'affectent pas la fréquence d'opération du circuit et ont un effet beaucoup plus négligeable sur la complexité. La profondeur des tampons du tunnel fait également varier sa complexité de façon importante. Le chemin critique du tunnel se situe lorsqu'il faut évaluer tous les paquets disponibles et faire le choix du paquet à envoyer lors du prochain cycle d'horloge. Lorsque le mode *retry* n'est pas supporté, la sélection du prochain paquet à envoyer est légèrement simplifiée.

Il existe plusieurs améliorations qu'il serait possible d'effectuer pour améliorer le tunnel, dont l'ajout d'utilisation de mémoires embarquées dans le module de *Contrôle de flux* et l'optimisation de l'utilisation des mémoires embarquées dans le *Tampon de commandes*. Ces modifications combinées pourraient permettrent d'économiser environ 20 % de la complexité du tunnel. Augmenter le nombre de bits traités à chaque cycle d'horloge permettrait d'augmenter de façon significative la bande passante du tunnel. Cette approche a été adoptée dans plusieurs solutions commercialement disponibles.

Chapitre 5. Intégration entre HT et un système embarqué

Les systèmes embarqués permettent d'intégrer un grand nombre de fonctionnalités à l'intérieur d'une puce tout en gardant un temps de développement raisonnable. Au cœur d'un système embarqué se trouve un système de communication qui permet aux divers composants de communiquer entre eux. De façon traditionnelle, un bus est utilisé à cet effet, mais plusieurs projets de recherche explorent également des topologies en réseau. Il y a par contre toujours une certaine limite au nombre de composants que l'on peut intégrer sur une même puce; il est donc parfois nécessaire de partitionner le système sur plusieurs puces. Il peut également être intéressant de distribuer un système sur plusieurs puces afin de le rendre extensible, en fonction des besoins de l'application utilisant le système.

Bien que plusieurs groupes de recherche se penchent sur la conception de technologies pour la communication dans les systèmes embarqués et que d'autres évaluent la façon de partitionner un système sur plusieurs puces [33], la méthode pour effectuer la communication entre divers systèmes embarqués distribués sur plusieurs puces n'a à notre connaissance pas été abordée dans la littérature. Un protocole de communication interpuces de haute performance, tel que HyperTransport, est idéal pour une telle tâche. Ce chapitre va donc se pencher sur la conception d'un pont permettant d'interconnecter des systèmes embarqués à une chaîne HT.

5.1 Opérations supportées par le pont

Pour être en mesure de faire la conception d'un pont entre HT et un système embarqué, il faut en premier lieu bien établir les divers types d'opérations supportées par le système embarqué. Il existe deux modèles de communication possibles : par l'échange de messages vers une destination précise ou des opérations d'écritures et de lectures de données à l'aide d'un système de mémoire partagée. Plusieurs bus de systèmes

embarqués tels que AMBA, OPB et OCP utilisent le modèle de mémoire partagée. C'est donc le modèle de mémoire partagé à l'aide d'opérations d'écritures et de lectures qui est supporté par le pont.

Pour effectuer des écritures et des lectures entre plusieurs systèmes embarqués, il est nécessaire d'échanger des paquets de types écriture et lecture. Dans le cas d'un système distribué sur plusieurs puces, la réponse à une lecture doit être acheminée à l'intérieur du réseau et il est donc également nécessaire d'utiliser des paquets de type réponse. Ces trois types de paquets sont supportés naturellement par HT, ce qui facilite grandement la tâche d'intégration. Il faut donc tout simplement faire la conception d'un pont qui permet de recevoir des requêtes provenant d'un système embarqué pour les acheminer à une chaîne HT. De la même façon, les requêtes provenant de la chaîne HT doivent être transformée en requêtes pour le système embarqué.

Les requêtes de lecture qui exigent une réponse causent par contre un certain problème à un pont puisqu'il n'est pas possible de tout simplement traduire une demande de lecture HT à une demande de lecture d'un système embarqué. Pour chaque lecture HT reçue, le pont doit mémoriser une variété d'informations sur cette lecture provenant de la chaîne HT pour permettre de générer une réponse HT appropriée lorsque le système embarqué répond à cette requête. De la même façon, lorsqu'une lecture provient du système embarqué, l'information sur l'origine de cette requête doit être emmagasinée jusqu'à ce qu'une réponse soit reçue de la part de la chaîne HT.

L'enregistrement de ces informations nécessite de l'espace mémoire, ce qui est limité à l'intérieur d'un circuit. Dans le cas où cet espace mémoire est entièrement utilisé, le tunnel doit refuser les opérations et attendre que des réponses soient reçues pour libérer de l'espace mémoire. Si la communication dans les deux sens est bloquée, on pourrait se retrouver en situation d'inter-blocage. Il est donc absolument nécessaire de s'assurer qu'il n'est pas possible qu'une requête de lecture puisse bloquer le passage

d'une réponse. HyperTransport prévoit déjà cette éventualité, mais cela n'est pas nécessairement le cas des systèmes embarqués qui utilisent des réseaux embarqués. Pour pallier à cette éventualité, le pont doit utiliser deux ports de communication avec le réseau embarqué : un qui ne traite que les requêtes et un deuxième qui ne traite que les réponses. L'utilisation de deux ports indépendants permet donc d'éviter les problèmes d'inter-blocage.

En général, lorsqu'il faut transférer une grande quantité de données entre des éléments d'un système embarqué, il est nécessaire d'effectuer plusieurs opérations sur le bus. Pour optimiser la communication, un bus permet souvent de transmettre plusieurs opérations en rafale. HT peut également bénéficier d'envoyer de l'information en rafale, puisque chaque paquet nécessite un en-tête : si plusieurs données partagent le même en-tête, cela réduit le trafic à l'intérieur de la chaîne HT. Le pont supporte également l'envoi et la réception de données en rafale. Le seul désavantage de l'envoi en rafale est que cela résulte en une latence accrue, puisque le pont va accumuler des données jusqu'à un maximum de 64 octets en provenance du système embarqué avant d'envoyer un paquet HT.

5.2 Structure du pont

Le pont a une structure très simple, puisqu'il ne s'agit que d'un traducteur de paquets. À cause de la non-disponibilité d'un outil de synthèse de SystemC au moment d'effectuer la conception du pont, il a été décidé d'utiliser le VHDL qui est un langage de description matérielle traditionnel. Le pont contient deux modules principaux : un pour traduire les paquets de HT vers le système embarqué et un deuxième pour traduire les transactions du système embarqué vers HT. Lorsque le pont reçoit des requêtes non supportées par le système embarqué, comme des écritures qui nécessitent une réponse après complétion, une réponse doit également être générée.

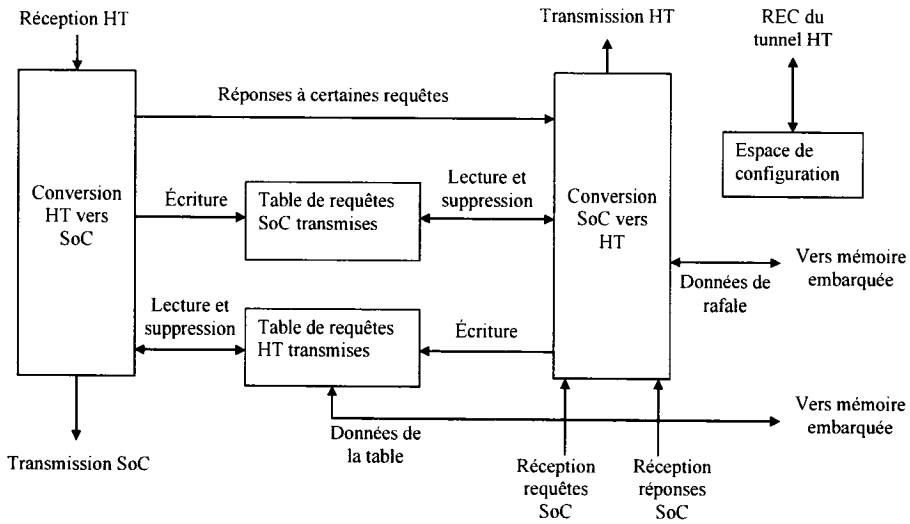

Figure 5.1 - Schéma bloc du pont HT – Système embarqué

Deux tables de conversion permettent d'enregistrer l'origine d'une lecture pour permettre de rediriger les réponses lorsqu'elles sont reçues : une table pour les lectures en provenance de HT et une autre pour les transactions en provenance du système embarqué. HT utilise un numéro unique pour identifier chaque transaction; les lectures en direction de HT peuvent donc être enregistrées à l'intérieur d'une mémoire embarquée et ensuite être récupérées facilement à l'aide de ce numéro unique. La table qui enregistre les informations sur les lectures en provenance du système embarqué doit par contre être contenue à l'intérieur de bascules puisque la récupération de ces informations est plus complexe. Une deuxième mémoire embarquée est par contre utilisée pour accumuler les données d'une requête en rafale en provenance du système embarqué.

5.3 Vérification du pont

Bien que le pont ait été conçu à l'aide du langage VHDL, contrairement au tunnel HT, le langage SystemC a quand même été utilisé pour en effectuer la vérification. Plusieurs outils permettent d'effectuer la co-simulation de SystemC et des langages de description matérielle traditionnels. SystemC fut utilisé à cause de l'aise avec laquelle il

est possible d'implémenter des structures complexes, particulièrement avec l'accès aux bibliothèques STL (*Standard Template Library*). Le banc d'essai du pont consiste en un générateur aléatoire de paquets et d'une seconde partie qui vérifie que les résultats produits par le pont sont valides.

5.4 Hôte de chaîne HT

Une chaîne de communication HT nécessite un élément HT de type hôte pour fonctionner correctement. L'hôte a le rôle d'initialiser les divers éléments de la chaîne et également d'effectuer une réflexion des paquets reçus. Le développement d'un circuit HT hôte serait un travail considérable considérant le travail qui a été requis pour faire la conception d'un tunnel, même si un grand nombre de composants du tunnel pourraient être réutilisés pour faire la conception de l'hôte. Afin d'accélérer l'intégration d'une chaîne HT, un hôte a été conçu de façon logicielle à un niveau d'abstraction transactionnel.

Pour être en mesure de faire un hôte à un haut niveau d'abstraction, il faut considérer que le tunnel fonctionne à l'aide d'un modèle très proche du matériel; il faut donc être en mesure de passer d'un niveau d'abstraction à un autre. Heureusement, les outils pour effectuer cette transition de niveau d'abstraction étaient déjà conçus dans le cadre de la vérification du fonctionnement du tunnel. Un schéma bloc de l'architecture logicielle de cet hôte est présenté à la figure 5.2. Une couche physique très proche du niveau matériel est sensible au signal d'horloge et s'occupe de faire l'initialisation de la communication et d'organiser les données reçues en vecteurs de 32 bits. Ces vecteurs de 32 bits sont transmis de façon transactionnelle par événement à une couche logique. La couche logique gère ces événements pour reconstituer les vecteurs reçus en paquets. Elle gère également le contrôle de flux et le mode *retry* si nécessaire. Lorsqu'un paquet est assemblé et validé, celui-ci est envoyé au modèle de l'hôte.

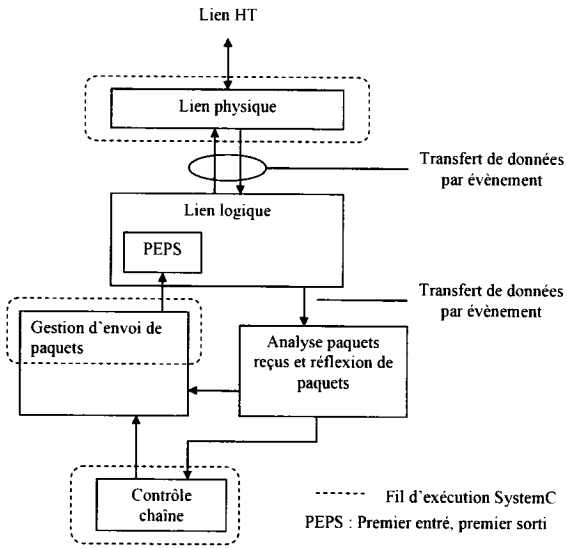

Figure 5.2 - Architecture de l'hôte HT (haut niveau d'abstraction)

Lorsqu'un paquet est reçu, celui-ci est analysé par le modèle de l'hôte et il est soit envoyé à la gestion de l'envoi de paquets afin d'y être réfléchi, ou il est transmis au contrôle de chaîne. Le module de gestion d'envoi de paquets s'assure de donner et libérer les numéros d'identification de paquets et de transmettre les paquets au lien logique. Le contrôle de chaîne génère des paquets pour effectuer la configuration des éléments de la chaîne.

5.5 Intégration des composants

Avec en main un tunnel HT, un hôte HT, un pont pour passer de HT à un système embarqué et un réseau embarqué, il est alors possible d'intégrer le tout à l'intérieur d'un système distribué sur plusieurs puces. Puisque certains éléments comme l'hôte HT sont logiciels, cette intégration n'est effectuée qu'au niveau d'une simulation. La présente section commence par une explication de la structure utilisée pour effectuer une démonstration de concept de l'intégration de tous ces composants et présente les résultats obtenus de cette intégration.

5.5.1 Démonstration de concept

À cause du temps limité à notre disposition, un système de complexité minimale fut conçu afin de démontrer le bon fonctionnement des divers composants. Le système comprend deux systèmes embarqués contenant chacun un microprocesseur et une mémoire embarquée. Puisque ceux-ci ont des interfaces OPB (On-chip Peripheral Bus), une enveloppe OPB est nécessaire pour leur permettre de communiquer avec le réseau embarqué RoC. Chacun des RoC est également connecté à un pont qui permet de relier le réseau embarqué à la chaîne HT. Dans le cadre de cette démonstration de concept, la gestion du numéro de RoC d'un paquet n'est pas effectuée par le réseau embarqué : un paquet en provenance du premier RoC est toujours dirigé vers le deuxième RoC et vice versa. Afin de faire la conception d'un système comportant plus de deux réseaux embarqués, il suffirait donc de modifier le réseau embarqué pour qu'il mémorise et dirige correctement un paquet qui contient un champ indiquant le numéro du RoC de la destination.

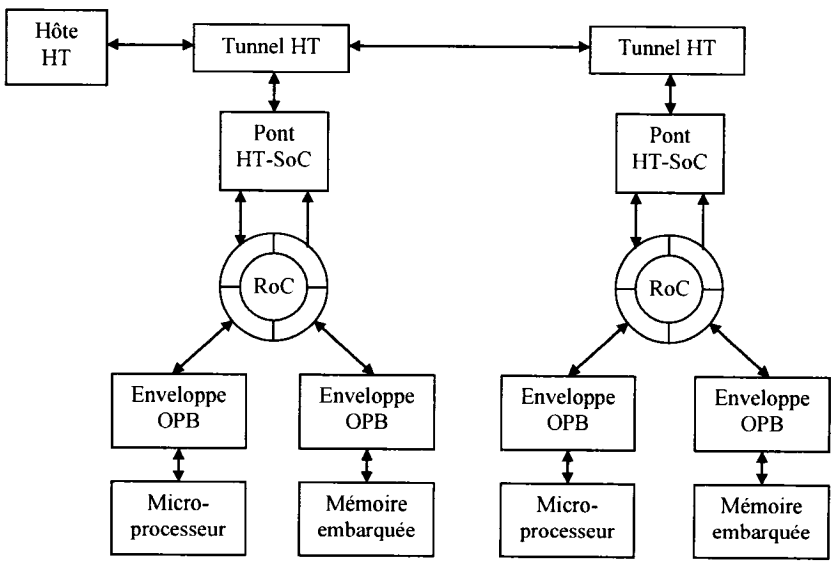

Figure 5.3 - Schéma bloc de l'intégration d'une chaîne HT et de réseaux embarqués RoC

5.5.2 Résultats

En simulation, le système présenté à la section précédente fonctionne sans faute. Les deux microprocesseurs peuvent accéder en lecture et en écriture aux mémoires embarquées de l'un ou l'autre réseau embarqué, et cela de façon transparente. À cause des limitations du système utilisé pour démontrer le concept, il n'est pas possible de tester le système avec une grande quantité de trafic. Il est néanmoins intéressant d'examiner la latence de communication qui est présentée dans le tableau suivant.

Tableau 8 - Latence moyenne des communications à l'intérieur de la chaîne HT

	Temps passé dans les RoCs (cycles)	Temps passé dans la chaîne HT (cycles)	Temps passé dans les ponts (cycles)	Temps total pour l'opération (cycles)
Envoi de paquet, DirectRoute non activé	48	25	2	75
Envoi de paquet, DirectRoute activé	48	13	2	63

Il est possible de remarquer que l'utilisation de la communication DirectRoute permet de réduire considérablement la latence de la communication. Ceci peut être expliqué par le fait que lorsque ce mode est activé, les paquets n'ont pas à être réfléchis par l'hôte de la chaîne comme le montre la figure 5.4.

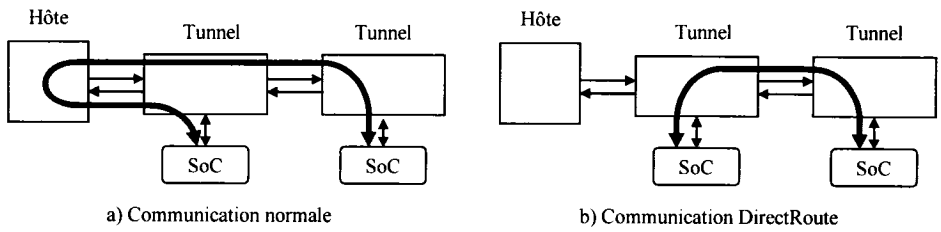

Figure 5.4 - Effet de DirectRoute sur la communication entre deux éléments d'une chaîne HT

Bien qu'il ne soit pas possible de faire la synthèse du système en entier parce que l'hôte de la chaîne est un élément logiciel, le pont, par contre, peut être synthétisé. Sa complexité varie en fonction de la taille de la table de correspondance utilisée pour emmagasiner les données des requêtes de lecture qui arrivent du réseau HT (voir la figure 5.5).

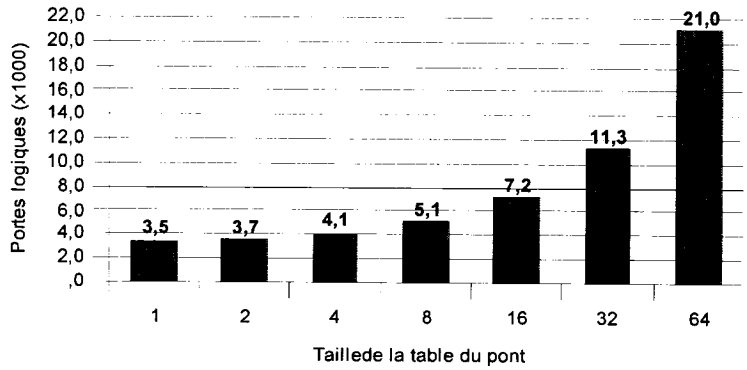

Figure 5.5 - Complexité du pont en fonction de la taille de la table qui enregistre la source d'une lecture HT

5.6 Comparaison de la complexité des divers éléments du système

Il est très difficile de bien saisir la complexité du tunnel HT sans avoir un point de comparaison avec d'autres types de circuits. Il devient donc intéressant de comparer le tunnel avec d'autres systèmes de communication tels que les réseaux embarqués, puisque ceux-ci permettent également le transfert de données par paquets. Une telle comparaison peut aussi permettre de bien comprendre d'où vient la différence de complexité entre ces systèmes. Commençons premièrement par présenter les données d'un RoC au tableau 9. Les résultats présentent un RoC où les paquets ont une taille de 83 bits, ce qui permet d'avoir des données de 32 bits, une adresse d'accès mémoire de 32 bits et plusieurs autres bits assurant la compatibilité avec le pont vers HT.

Tableau 9 - Comparaison de résultats de synthèse des nœuds élémentaires pour les réseaux HT et RoC

Système	Complexité - FPGA Xilinx (LUT)
Tunnel avec options par défaut	18 100
RoC avec paquet de 83 bits et 4 ports	975

Il existe une très grande différence entre la complexité d'un système de communication HT et un réseau embarqué RoC, surtout si l'on considère que les données dans la table précédente sont pour un RoC qui comporte 4 ports tandis que le tunnel ne suffit qu'à la communication d'un seul composant. Il y a plusieurs explications de cette grande différence de complexité :

- HT contient plusieurs canaux virtuels, ce qui implique qu'il faut de la logique pour effectuer la sélection de paquets à envoyer ainsi que des tampons pour emmagasiner ces paquets. Sans canaux virtuels, il n'y a pas de sélection à faire et des mémoires embarquées peuvent servir de tampon, réduisant grandement la complexité logique du circuit. La somme des tampons de paquets avec la logique de gestion de ceux-ci occupent 44,0 % de l'espace total du tunnel.

- Puisque la communication interpuces comporte des risques de corruption de données, de la vérification d'intégrité des données et de la correction d'erreurs sont nécessaires. L'ensemble des modules de calcul de CRC et d'historique des paquets occupe 18,5 % du tunnel.

- Chaque élément d'une chaîne HT est relativement indépendant, avec son espace de configuration, tandis qu'un réseau embarqué ne nécessite pas un tel espace de configuration. Le module REC qui contient cet espace registre

occupe 6,6 % de l'espace du tunnel, sans compter la logique nécessaire pour gérer de lui faire parvenir et traiter des paquets.

- Les paquets HT n'ont pas tous la même taille et les en-têtes sont séparés des données, ce qui nécessite de gérer l'assemblage des en-têtes et l'enregistrement des données. Plus tard, il faut faire l'inverse pour envoyer les paquets. Toute cette logique occupe plus de 7 % du tunnel.

- Pour éviter les débordements de capacité, un système de gestion de contrôle de flux est nécessaire dans HT, ce qui occupe 3,3 % du tunnel. *Note : Le contrôle de flux du Tampon de données n'est pas inclus parce que sa complexité est comprise dans le point précédent.*

- Puisque chaque élément est indépendant, il est également nécessaire d'initialiser la communication, ce qui occupe 3,2 % de la logique du tunnel.

- HT comporte un algorithme d'équité afin d'empêcher qu'un seul élément de la chaîne ne monopolise la communication. La mise en œuvre de cet algorithme occupe 2,9 % du tunnel.

En plus de tous ces facteurs précis, la richesse fonctionnelle de HT fait en sorte qu'une grande quantité de logique de contrôle est nécessaire pour gérer toutes ces fonctionnalités.

5.7 Conclusion

La conception d'un pont qui permet de lier un système embarqué à une chaîne HT a permis de faire une démonstration de concept qui intègre deux réseaux embarqués RoC et un réseau de communication HT. Le pont permet de traduire les paquets HT en écritures, lectures et réponses telles que supportées par la majorité des systèmes embarqués. À cette fin, le pont contient une table qui lui permet d'emmagasiner l'origine des requêtes

pour permettre de bien rediriger les réponses à ces requêtes lorsque celles-ci sont reçues. Le pont a été vérifié à l'aide d'un banc d'essai en SystemC.

Puisqu'une chaîne HT nécessite un hôte pour initialiser la chaîne et effectuer de la réflexion de paquets, un hôte a dû être conçu. Pour accélérer le temps de développement, l'hôte a été implémenté à un niveau d'abstraction très élevé. Cet hôte, intégré avec deux tunnels HT, deux ponts et deux RoCs contenant chacun une mémoire et un microprocesseur, ont pu démontrer le bon fonctionnement du système. Tel qu'il est présentement, le système ne permet d'intégrer que deux réseaux embarqués, mais cela résulte tout simplement d'un manque de temps pour développer les enveloppes nécessaires pour supporter un plus gros réseau, et non pas d'une limitation technique.

Le temps requis pour passer d'un réseau embarqué à un autre à l'aide de la chaîne HT est de 25 cycles de façon normale et de 13 cycles avec l'activation de DirectRoute. L'intégration du réseau RoC avec un réseau HT est une bonne occasion de comparer les deux technologies. Un seul tunnel HT occupe plus de 18 fois l'espace d'un RoC à 4 ports. La raison principale de cette différence de complexité provient du fait que HT a plusieurs canaux virtuels et nécessite donc beaucoup de logique pour effectuer une bonne sélection des paquets à transmettre. D'autres facteurs incluent la vérification et la correction d'erreurs, la présence d'un espace de configuration dans le tunnel, la nécessité de gérer l'assemblage et l'enregistrement des paquets, le contrôle de flux, l'initialisation de la communication, l'algorithme d'équité et finalement toute la logique associée à gérer la richesse fonctionnelle de HT.

Conclusion

Les besoins toujours plus grands de calcul de la part d'une multitude d'applications demandent souvent de partitionner la fonctionnalité d'un système sur plusieurs puces. Pour répondre à ce besoin, un système intégrant le protocole de communication interpuces HT avec des systèmes embarqués à l'aide d'un pont a été présenté. Puisque peu d'informations sont disponibles sur la conception et la complexité des circuits qui gèrent de la communication interpuces de haute performance, il existe un grand intérêt à faire la conception et à analyser les éléments qui contribuent à la complexité d'un tunnel HT. Le pont proposé pour passer d'un réseau embarqué à HT est innovateur du fait qu'il permet d'intégrer une grande variété de systèmes embarqués de façon transparente.

La conception d'un tunnel HT est une très grande composante de ce mémoire à cause de la grande complexité du protocole. Le tunnel a été conçu à l'aide du langage SystemC et est entièrement synthétisable en plus d'être paramétrable. Le tunnel est divisé en 8 grands modules : *Lien* qui gère l'initialisation de la communication, *Tampon de données* qui accumule les en-têtes des paquets reçus, *Tampon de commandes* qui accumule les données reçues, *Gestion erreur* qui traite les paquets reçus par erreur, *Interface* qui permet au tunnel de communiquer avec un utilisateur, *REC* qui contient une multitude de registres pour afficher et configurer dynamiquement les paramètres du tunnel et finalement *Contrôle de flux* qui gère les paquets à envoyer.

Une très grande part du temps de conception d'un circuit est nécessaire pour effectuer une vérification que la fonctionnalité de ce circuit est correcte. Une approche de vérification ascendante a été utilisée pour la vérification du tunnel : une multitude de bancs d'essai C++ utilisant la bibliothèque SystemC permettent de vérifier le bon fonctionnement d'une grande partie des sous-modules, des modules et finalement du design en entier. Le tunnel est générique et peut être synthétisé à la fois pour la

technologie de cellules normalisées CMOS 0,18 µm de TSMC qui permet d'obtenir une bande passante minimale de 8000 Mb/s et pour les FPGA tels que le Stratix (-5) d'Altera et Virtex II Pro (-7) de Xilinx avec une bande passante de 2400 Mb/s et 3200 Mb/s respectivement. Le tunnel requiert un total de 43,2 Kibits de mémoire embarquée, mais afin de demeurer générique, le tunnel n'inclut pas ces mémoires; il utilise plutôt des ports pour y accéder.

Le chemin critique du tunnel se situe au niveau de la sélection du paquet à envoyer vers le lien physique à partir des diverses sources de paquets. La fréquence d'opération varie de 88,4 MHz pour un FPGA d'Altera jusqu'à 251,3 MHz pour la technologie de cellules normalisées. La bande passante du tunnel est déterminée à la fois par la fréquence d'opération du circuit et la quantité de données traitées à chaque cycle d'horloge. Le tunnel traite 32 bits par cycle d'horloge et, pour augmenter la bande passante du tunnel, il serait donc nécessaire de le modifier afin qu'il traite 64 ou même 128 bits par cycle d'horloge.

La complexité logique équivalente du tunnel se situe à 131,9 milliers de portes logiques, ce qui exclut les mémoires embarquées. Un total de 44 % de l'espace du tunnel est occupé par des registres qui permettent d'accumuler des paquets et de les réorganiser. Le fait que HT utilise plusieurs canaux virtuels fait considérablement augmenter la complexité des tampons, puisqu'il faut faire la comparaison des paquets des différents canaux virtuels et, pour faire cela, il faut accumuler les paquets à l'intérieur de bascules au lieu de garder la totalité des paquets à l'intérieur de mémoires embarquées. Une utilisation plus judicieuse de mémoires embarquées au niveau du *Tampon de commandes* et du *Contrôle de flux* permettrait de couper la complexité du tunnel d'environ 20 %.

Malgré le fait que ce sont les tampons qui occupent une majorité de l'espace du tunnel, plusieurs autres facteurs demandent une grande quantité de ressources. Les mécanismes de vérification et de correction d'erreurs occupent 18,5 % du tunnel. En

plus, ce dernier chiffre ne compte pas une certaine partie de la logique de contrôle du mode *retry*, qui permet de corriger des erreurs, puisque celle-ci est fortement intégrée à plusieurs endroits dans le tunnel. Lorsque ce mode est désactivé lors de la synthèse, il est possible d'économiser 15,8 % des ressources requises pour la mise en œuvre du tunnel. D'autres éléments qui occupent une partie de l'espace du tunnel sont les registres d'espace de configuration (*REC*) à 6,6 %, la gestion des en-têtes des paquets et des paquets de données à 7 % du tunnel, le contrôle de flux à 3.3 %, un système d'initialisation de la communication à 3,2 % et un algorithme d'équité d'insertion de paquet à 2,9 %. Un lecteur averti pourra noter que la somme de ces fonctionnalités ne totalise pas 100 % et cela est normal, puisqu'une certaine quantité de logique est nécessaire pour contrôler toute cette richesse fonctionnelle.

Un aspect très intéressant du tunnel est sa multitude de paramètres. L'option de supporter ou non le mode *retry* est l'option ayant le plus d'impact sur le tunnel. D'autres options disponibles incluent la possibilité de faire l'alignement des données à l'interne du tunnel et le support du mode DirectRoute, qui n'ont tous deux qu'un très petit impact sur la complexité du tunnel. La profondeur du *Tampon de commandes* peut également être contrôlée, ce qui fait varier de façon considérable la taille du tunnel. Faire passer la profondeur du *Tampon de commandes* de 3 à 15 paquets résulte en une augmentation de complexité de 25,2 milliers de portes logiques. Il est également possible de contrôler si les paquets sont réorganisés à l'intérieur des tampons, ce qui permet de couper une fraction considérable de la complexité du *Tampon de commandes.* Il est également possible de contrôler la profondeur des modules *Tampon de données*, mais cela affecte plutôt la quantité de mémoire embarquée requise et a un effet négligeable sur la complexité logique du tunnel. Un dernier paramètre qui a un énorme effet sur la complexité du tunnel est la profondeur des tampons de paquets en provenance de l'utilisateur, qui fait augmenter la complexité logique du tunnel de 16,3 milliers de portes logiques lorsque l'on passe d'une profondeur de 2 à une profondeur de 6. Ce tampon est

nécessaire pour absorber la latence de la communication avec l'utilisateur et celui-ci pourrait dans le futur fortement bénéficier de l'utilisation de mémoire embarquée.

Le tunnel a ensuite été intégré à un réseau embarqué à l'aide d'un pont. Ce pont permet de traduire les paquets HT en écritures, lectures et réponses, tels qu'ils sont supportés par la majorité des systèmes embarqués. Puisqu'une chaîne HT nécessite un hôte pour initialiser la chaîne et effectuer de la réflexion de paquets, un hôte a également été conçu à un niveau d'abstraction très élevé. Cet hôte, intégré avec deux tunnels HT, deux ponts avec deux réseaux embarqués RoCs, contenant chacun une mémoire et un microprocesseur, ont pu démontrer le bon fonctionnement du système.

Ce mémoire de maîtrise a donc réussi à présenter un système permettant d'intégrer un réseau de communication interpuces à des systèmes embarqués, tout en apportant une multitude d'informations sur les protocoles de communication haute performance tel que HyperTransport. En plus de fournir ces informations, le tunnel est librement disponible pour la communauté scientifique sous une licence Mozilla Public Licence (MPL).

Travaux futurs

Les travaux effectués dans le cadre de ce mémoire ont permis de créer un système de communication complexe, mais uniquement en simulations. Une implémentation matérielle intégrant plusieurs réseaux embarqués à l'aide de tunnels et d'un hôte HyperTransport permettrait d'évaluer la performance réelle d'un tel système. Afin d'obtenir une implémentation matérielle, il sera en premier lieu nécessaire d'avoir accès à de nouveaux outils afin d'effectuer la synthèse du tunnel.

Un système complexe nécessite beaucoup plus que des tunnels HT : il nécessite des commutateurs afin d'optimiser la topologie du réseau. Puisqu'un tunnel HT est en fait un commutateur HT à deux ports, plusieurs éléments du tunnel pourraient être utilisés afin de développer un commutateur HT à plus de deux ports. Il serait également bénéfique

qu'un système puisse garantir une certaine performance, ce que HyperTransport n'offre pas. Afin de garantir la performance d'un système de communication complexe, il serait donc nécessaire d'implémenter des algorithmes de partage de temps dans le réseau ou d'aller jusqu'à utiliser un réseau parallèle à HT avec des liaisons commutées.

Références

[1] ALTERA CORPORATION, "HyperTransport MegaCore User Guide", Doc. : ug_hypertransport.pdf, http://www.altera.com, octobre 2005.

[2] CELOXICA, « Agility Compiler Product Brief », Doc. : CEL-W06215E4L-335.pdf, http://www.celoxica.com/products/agility, 2006.

[3] C. GRECU, P. PRATIM PANDE, A. IVANOV et R. SALEH, « A Scalable Communication-Centric SoC Interconnect Architecture », ISQED2004, p. 343-348, juin 2004.

[4] C.N. KELTCHER, K.J. MCGRATH, A. AHMED et P. CONWAY, « The AMD Opteron processor for multiprocessor servers » Micro IEEE, p. 66-76, mars-avril 2003.

[5] CRAY INC., « Cray XT3 Datasheet », Doc. : Cray_XT3_Datasheet, 2005.

[6] C. SAUER, M. GRIES, J. IGNACIO GOMEZ, S. WEBER et K. KEUTZER, « Developing a Flexible Interface for RapidIO, Hypertransport, and PCI-Express », PARELEC'04, p. 129-134, septembre 2004

[7] D. C. PHAM et al., « Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor», IEEE Journal of solid-state circuits, vol. 41, n° 1, p. 179-196, janvier 2006

[8] E. GRIMPE et OPPENHEIMER, « Extending the SystemC Synthesis Subset by Object-Oriented Features », CODES+ISSS'03, p. 25-30, octobre 2003.

[9] E. RIJPKEMA, K. GOOSSENS, A. RADULESCU, J. DIELISSEN, J. VAN MEERBERGEN, P. WIELAGE et E. WATERLANDER, « Trade-offs in the design

of a router with both guaranteed and best-effort services for networks on chip, Computers and Digital Techniques », IEE Proceedings-Volume 150, Issue 5, 22 sept. 2003, p. 294–302.

[10] FORTE, « Cynthesizer Provides the Fastest Path to Silicon », http://www.forteds.com/products/cynthesizer.asp, 2006.

[11] F. ST-PIERRE, mémoire. « Modélisation à bas niveau et analyse d'un réseau intégré sur puce », École Polytechnique de Montréal, (mémoire en préparation), 2006.

[12] GDA TECHNOLOGIES, INC., « HyperTransport Cave Core», Doc : gda_ht_cave.pdf, http://www.xilinx.com, septembre 2003.

[13] GDA TECHNOLOGIES, INC., « HyperTransport™ Cave», Doc. : gda_hypercave.pdf, http://www.xilinx.com, mars 2003.

[14] GDA TECHNOLOGIES, Inc., « HyperTransport Tunnel», Doc. : GHT-Tunnel.pdf, http://www.gdatech.com/IP/pdf.shtml, janvier 2004.

[15] G. KEES et al., chapitre 4 : « GUARANTEEING THE QUALITY OF SERVICES IN NETWORKS ON CHIP », A. Jantsch et H. Tenhunen, éditeurs, « Networks on Chip », p. 61-82, Kluwer Academic Publishers, mars 2003.

[16] HYPERTRANSPORT TECHNOLOGY CONSORTIUM, « Hypertransport I/O Link Specification Rev. 2.00b », Doc. : HTC20031217-0036-0009, avril 2005.

[17] HYPERTRANSPORT CONSORTIUM, « HyperTransport™ EATX Motherboard/ Daughtercard Specification », Doc. : HTC2004105-0040-0006, octobre 2005.

[18] JEAN-FRANÇOIS BÉLANGER, mémoire : « Développement d'un module de gestion d'envoi pour un tunnel HyperTransport™ », juillet 2004.

[19] L. CHAREST et P. MARQUET, « Comparisons of Different Approaches of Realizing IP Block Configuration in SystemC », NEWCAS2005, p. 83-86, juin 2005.

[20] MARTTI FORSELL, A Scalable High-Performance Computing Solution For Networks on Chips, IEEE Micro, p. 46–55, septembre-octobre 2002.

[21] M. BEDFORD TAYLOR, W. LEE, S. AMARASINGHE et A. AGARWAL, « Scalar operand networks: on-chip interconnect for ILP in partitioned architectures », HPCA-9 2003, p. 341-353, février 2003.

[22] MOZILLA FOUNDATION, « The Mozilla Public License, version 1.1. », http://www.mozilla.org/MPL/.

[23] O. HÉBERT, I.C. KRALJIC et Y. SAVARIA, « A Method To Derive Application-Specific Embedded Processing Cores », CODES 2000, p. 88-92, mai 2000.

[24] OPENCORES.ORG, « Opencore coding guidelines », juillet 2003, http://www.opencores.org/cvsget.cgi/common/opencores_coding_guidelines.pdf.

[25] OPTICAL INTERCONNECT FORUM, « System Packet Interface Level 4 (SPI-4) Phase 2 Revision 1: OC-192 System Interface for Physical and Link Layer Devices », Doc. : OIF-SPI4-2.01.pdf, octobre 2003.

[26] P. GUERRIER et A. GREINER, « A Generic Architecture for On-Chip Packet-Switched Interconnections », DATE2000, p. 250-256, mars 2000.

[27] P.T. WOLKOTTE, G.J.M. SMIT, G.K. RAUWERDA et L.T. SMIT, « An Energy-Efficient Reconfigurable Circuit-Switched Network-on-Chip », IPDPS2005, p. 1025-1040, avril 2005.

[28] RAPIDIO TRADE ASSOCIATION, « RapidIO specification 1.3 », http://www.rapidio.org/specs/current, juin 2005.

[29] S. MURALI et G. DE MICHELI, « An Application-Specific Design Methodology for STbus Crossbar Generation », DATE'05, p. 1176-1181, mars 2005.

[30] S.R. HASAN, A. LANDRY, Y. SAVARIA et M. NEKILI, « Design constraints of a hypertransport-compatible network-on-chip », NEWCAS2004, p. 269-272, juin 2004.

[31] SYSTEMCRAFTER, « Cynthesizer SystemCrafter SC 2.0 », http://www.systemcrafter.com/index.php?page=products&product=sc, 2005.

[32] The OPEN SYSTEMC INITIATIVE (OSCI), « SystemC v2.0.1 functional specification », Doc. : SystemC_2.0_functional_spec, avril 2002.

[33] Y. FEI et N. K. JHA, « Functional Partitioning for Low Power Distributed Systems of Systems-on-a-chip », VLSID'02, p. 274-281, mars 2002.

[34] XILINX, « HyperTransport Single-Ended Slave (HTSES) Core v2.1 », Doc. : hypertransport_ds.pdf, http://www.xilinx.com, octobre 2003.